NetWarriors™ in C++

Programming Multiplayer Games for Windows®

CD-ROM Includes:

- Complete multiplayer game with modem, ethernet, and Internet support
- Software for better games graphics
- Game design tools
- Tons of C++ code

Joe Gradecki

NE

Prog

NetWarriors in C++

Programming Multiplayer Games for Windows

Joe Gradecki

Wiley Computer Publishing

WILEY

John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Katherine Schowalter
Editor: Tim Ryan
Managing Editor: Micheline Frederick
Text Design & Composition: Pronto Design & Production Inc.

Library of Congress Cataloging-in-Publication Data:
Gradecki, Joe. 1967-
NetWarriors in C++: programming multiplayer games for Windows / Joe Gradecki.
p. cm.
Includes index.
ISBN 0-471-11337-9 (pbk.)
1. Computer games—Programming. 2. C++ (Computer program language). 3. Microsoft Windows (Computer file)
I. Title.
QA76.76.C672G73 1996 95-46040
794.8'152—dc20 CIP

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Contents

Introduction

Networks, and especially the Internet, are growing at an exponential rate. One of the quickest growing areas is programming and playing games over the Internet. Netwarriors in C++ will get you into the driver's seat by showing you how to design and develop network games that allow up to four players to play over the Internet. To accomplish this we'll be using Visual C++, one of the most advanced programming environments available for Windows. Many tools and techniques are available to the programmer in this environment that allow for quick development of multiplayer graphical games.

Our Internet connection will be through SOCKETS, a mechanism made popular in the UNIX operating system. Microsoft Winsock will be our gateway to sockets in the Windows environment. Throughout the book, we will show the steps necessary for developing a game called King's Reign. This game allows up to four people to connect and play over the Internet in real time. Players are in control of an army of calvary, archers, legions, and flying dragons that they purchase, move strategically around the playing field, and use to defend their own castles while attacking their enemies. A sophisticated war engine that we build in the course of this book judges the relative strength of the units involved and then determines the outcome of battle. I'll teach you the techniques you need and show you how to create this game from the ground up, one line of code at a time.

CDROM

Enclosed on the CD-ROM, you will find both source code and executables for King's Reign, a multiplayer war game for up to 4 players that works over modems, networks, and the Internet. See Appendix A for details on playing the game.

You'll also get a powerful collection of paint programs, icon developers, sound utilities, Internet utilities, source code, and technical documentation that will help you design and build your own multiplayer Windows Internet game.

In the Book

Chapter 1 is an introduction to writing network based games. The focus of this chapter is to describe the concept of a network game and all of the different components necessary to develop one.

Chapter 2 sets the stage for the remainder of the book. Based on the topics discussed in Chapter 1, this chapter gives the design of King's Reign, a Windows multiplayer network game—what it is and how it is supposed to work.

Chapter 3 describes how the Microsoft Windows system handles graphics. Examples are created using C and the Microsoft Development Kit.

Chapter 4 expands on the concepts from Chapter 3. It discusses the Visual C++ environment and shows how to do graphics in C++ using Microsoft Foundation Classes.

Chapter 5 discusses the development of the foundation to our game, including the main window and its class, the bitmap background, sprites, and mouse messages.

Chapter 6 advances the creation of our game by adding an operations window to the game as well as a buy_equipment window.

Chapter 7 integrates the game foundation and the game operations support window to create a complete single player game.

Chapter 8 discusses sockets; what they are, where they come from, and how they are programmed.

Chapter 9 builds from Chapter 8 and discusses the Microsoft Windows version of sockets called Winsock. It discusses the sockets and any differences from traditional sockets as well as how to develop a simple communication program using sockets.

Chapter 10 begins with the design for integrating sockets into our game and ends with the actual implementation of full multiplayer functionality.

Chapter 11 shows how to extend the system to use a modem between two players.

Chapter 12 concludes the book with a discussion on adding support for Windows for Workgroups.

Appendix A describes how to play King's Reign.

Appendix B gives the complete code listing for the multiplayer version of King's Reign.

Chapter 1

Writing Network Games

The latest craze in computer games is multiplayer games. Modem games have been around for quite some time, but they only allow two players. In the last two years, several games have been introduced to the shareware and commercial markets that allow network playing. Games like DOOM and Descent include support for playing with four people over a network. This book will show you how to create a game that allows the same amount of interaction between four players; the kind of gaming interaction that players really want.

Concepts

When we speak about network games, we mean games that will allow some number of people to interact with each other in the same environment over a network, which can be anything from two computers connected with ethernet cards to something as large and complex as the Internet. Let's look at several things in that description.

Number of People

For many years, computer games have been available that support two people over a modem. It's fun to play with another person, but being able to play with four players would be even better. Increasing the number of players to four does cause a problem for the game though, because it has to do a lot more work to track players' information and communicate it to the other players. A game can bog down quickly, and a slow game is no game at all.

Look at how a game with two players works. When you connect to another player, every move you make is sent to the other player and all of the moves of the other player are sent to you. The game application only has one channel to monitor and buffer if necessary. When the application is changed so that four players can play, it will have to monitor three channels to all of the other players. Not only are there three channels, there are three buffers, and each move by each player will have to be sent to all other players simultaneously. That certainly is a large amount of information flowing between players.

In addition to the channel problem, we also have a problem of determining which player did what action at what time. Let's look at an example. In a game, player 1 has engaged in a war with player 3 and at the same time player 3 has engaged in a war with player 4. Can you see the problem? Player 3 thinks that he or she has a healthy playing piece to engage in war with player 4. But player 1 is currently in a war with player 3. Player 1's computer will perform the war between player 3 and player 1 and send the new health value of player 3 to player 3. Has player 3's machine performed the war with player 4? If so, several different values for player 3's playing piece will be floating in the network.

You might ask, "Why don't we just use the clocks on each machine to determine who engaged player 3's playing piece first?" The answer is there is no easy way to synchronize all four clocks. This is actually a common problem in the theory of distributed systems. There are several solutions, but they take more processing power than we want to dedicate to this problem.

The solution we will use is to allow all players to move simultaneously, buy new units, assess their units' damage, and so on, but only allow one player to declare war at a time. To accomplish this, we'll pass a "war" token among the players that determines who can declare war when. While this might sound limiting, it works very well. During tests of the software, there was hardly ever a situation where two people wanted to a start a war at the same time. In fact, many times the players were trying to outrun a persuing army.

Interacting

For a game to be fun to play it should have a certain level of interaction. What do the players do in the game? Our game is based on a turf war. Up to four warlords will battle for control of a large battlefield.

Although there is a token indicating who is allowed to initiate a war, all players can still move playing pieces around the board, purchase new playing pieces, and create a strategy for winning. Experience has shown that two players engaged in war will be so preoccupied that the other players will be able to set things up for a winning move.

In addition to the interaction on the playing field, network games should allow for communication to be available. Whether it's a messaging system or some other technique, all of the players should be able to talk with each other.

Same Environment

This might *seem* obvious, but all of the players in the game should be in the same environment. If a player moves a playing piece, all players should see the movement on their own screens. The visual movement must be accomplished relatively quickly, so that all of the players' screens are in sync.

Windows and Networking

When Microsoft introduced the Windows for Workgroups operating environment, its major feature was the ability to connect computers in a peer-to-peer fashion. If set up for it, each computer has access to other computers, which greatly increases the sharing of information. This was an important step for networking because the cost of connecting computers together was minimal.

Windows Games

When Microsoft introduced its networking operating environment Windows for Workgroups 3.11, it saw the need for a network game and included

Hearts. Hearts can be played by up to four players. One of the players is the dealer, and the rest are players. As we will see in Chapter 8, a dealer/player combination will be called a server/client setup.

What You Need

In order to use the information that you will find in this book, you need to have some knowledge of C++ and Windows. It is really not important to have programmed in Windows, because we are going to take a slow and visual look at just how everything works together and how to create this game one line of code at a time.

The software we create for the game will be a mix of C++ and C code. The code will be compiled using Microsoft Visual C++. The majority of the code will be written for the Microsoft Foundation Classes or MFC, a library that includes complete classes for doing most of the tedious things a Windows application must do.

Additional control for the program will come from C++ code that we add. The C++ code will be augmented with C code for such things as socket communication support and modem control. The final code will be designed to execute on any version of Windows including Windows NT and Windows 95. You will be on the cutting edge of technology, so grab your computer and your compiler and let's look at some code.

Chapter 2

THE GAME KING'S REIGN

The main purpose of this book is to teach you how to design and develop a network game using C++ and Microsoft Windows. To do this effectively, we will document how the game on the CD-ROM accompanying this book was developed. The game is called King's Reign. This chapter begins by giving you a textual description of the game and game play (see Appendix A for a complete set of instructions for playing the game). You will get a detailed understanding of the operations the game must handle. After the description, a design is laid out that discusses the objects used in the game and the interactions between the objects. The chapter ends with a discussion of the communication paths used in the game. With any software development project, you should begin with a set of specifications. The specifications will tell you throughout the project what the code is suppose to do. As I have already developed the game, the specification for you will be the description of the game and its accompanying illustrations. Once the specifications are set, we can put down a design for the project. In our case, the design is the objects and their interactions.

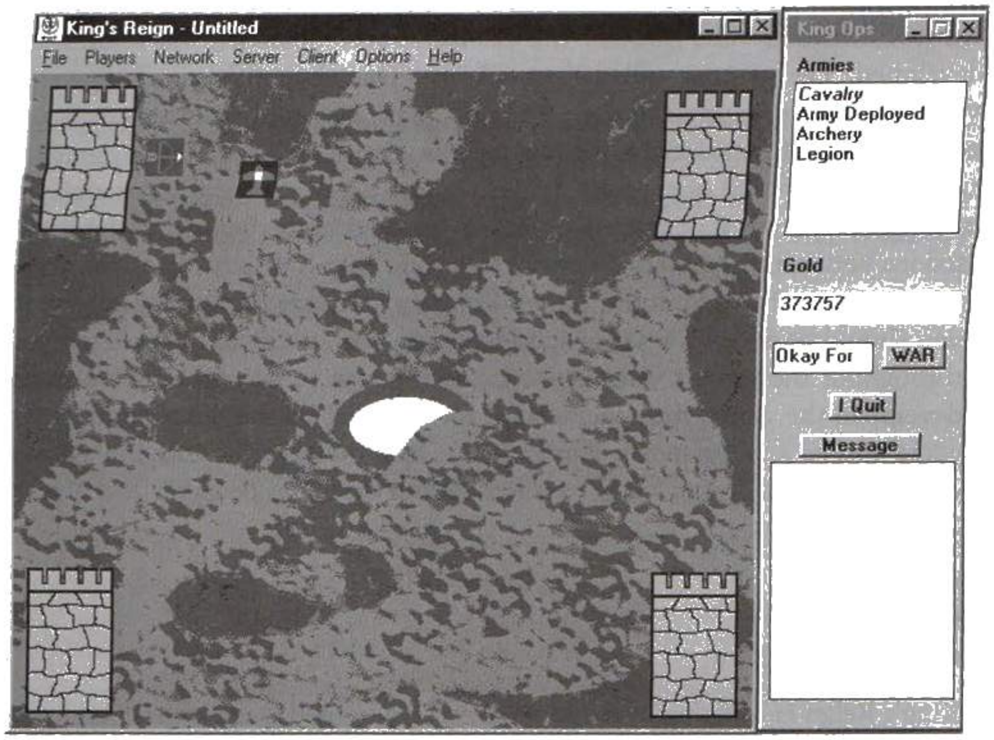
King's Reign

King's Reign is a multiplayer network game designed using objects and programmed in Visual C++ 1.5x. It will execute on any Microsoft Windows platform, including versions 3.1x and the new Windows 95 system. The game is set in medieval times with a single playing field and four different playing pieces. Figure 2.1 shows a game in progress.

The game allows from two to four players to engage in battle. The interconnections between players are performed using either a modem or the Internet. Obviously, if you are using the modem, you are limited to only two players. If you are using the Internet, you can connect the full four players. It should be noted that the game will allow the use of PPP dial-up lines to the Internet as well. Once all players are connected, they can purchase playing pieces. Figure 2.2 shows the dialog box used in the game to buy pieces.

The pieces available are:

- *Cavalry:* designed as a playing piece that moves via horses and engages in hand-to-hand combat. The cavalry has fairly quick

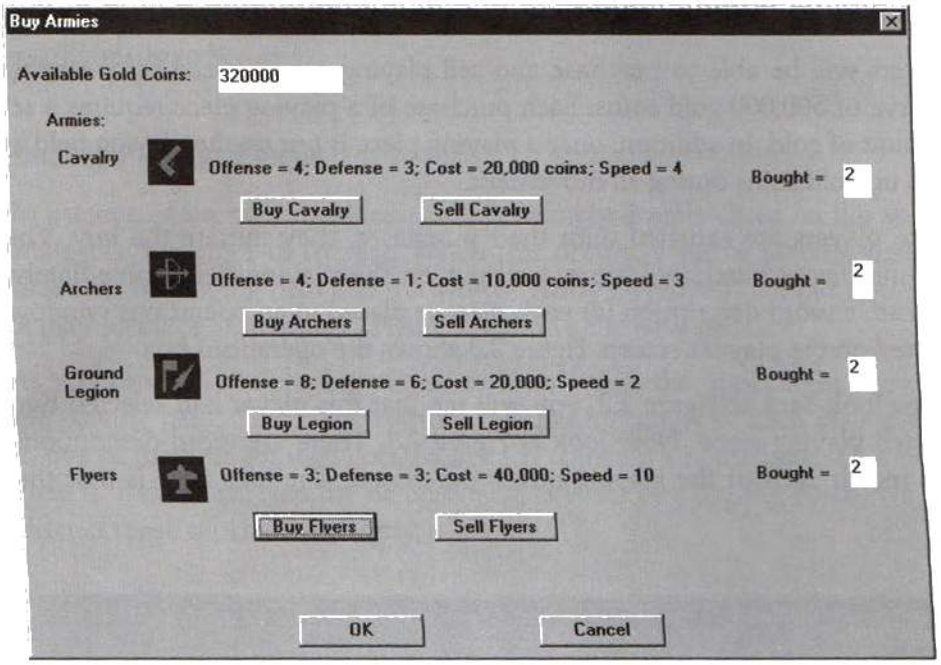


Cavalry
Army Deployed
Archery
Legion

373757

Okay For

Figure 2.1 *A King's Reign game in progress.*


Buy Armies

Available Gold Coins: 320000

Armies:

Cavalry — Offense = 4; Defense = 3; Cost = 20,000 coins; Speed = 4 — Bought = 2

Archers — Offense = 4; Defense = 1; Cost = 10,000 coins; Speed = 3 — Bought = 2

Ground Legion — Offense = 8; Defense = 6; Cost = 20,000; Speed = 2 — Bought = 2

Flyers — Offense = 3; Defense = 3; Cost = 40,000; Speed = 10 — Bought = 2

Figure 2.2 *The King's Reign dialog box for purchasing playing pieces.*

movement around the playing field and has average offensive and defensive scores.

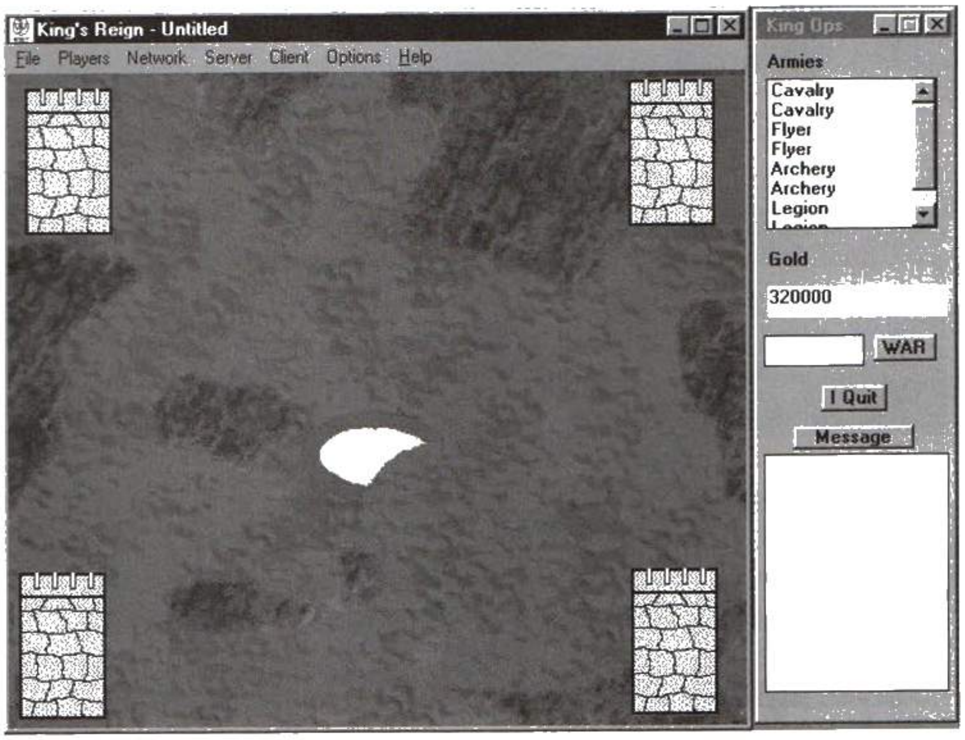
- *Legion:* designed as a playing piece that moves via foot and uses both hand-to-hand combat as well as primitive weapons. The legion is slow-moving because of its size. To offset its slowness, the legion is an offensive monster.
- *Flyers:* designed as a playing piece that moves very quickly on the playing field. The flyer is a man unit that uses a form of air travel. The flyers have high defense points, but low offense.
- *Archers:* designed as a playing piece that moves at an average tempo around the field. The archers are man units that use bow and arrows as their main weapon. Archers are a tough unit with high offense points but being without horses, low defense points.

Purchasing Playing Pieces

Players will be able to purchase and sell playing pieces based on a starting reserve of 500,000 gold coins. Each purchase of a playing piece requires a set amount of gold. In addition, once a playing piece is put on the playing field, it uses up gold coins during all movement.

Once players are satisfied with their purchases, they initiate the buy. The playing pieces purchased are not all put on the playing field immediately. Instead, a word description for each piece is placed in an operations window located on the player's screen. Figure 2.3 shows the operations box.

If you look back at Figure 2.2, you will see that this player had selected two of each playing piece. Now look at Figure 2.3. There are word descriptions that match each of the playing pieces purchased. The idea here is that the



Cavalry
Cavalry
Flyer
Flyer
Archery
Archery
Legion

320000

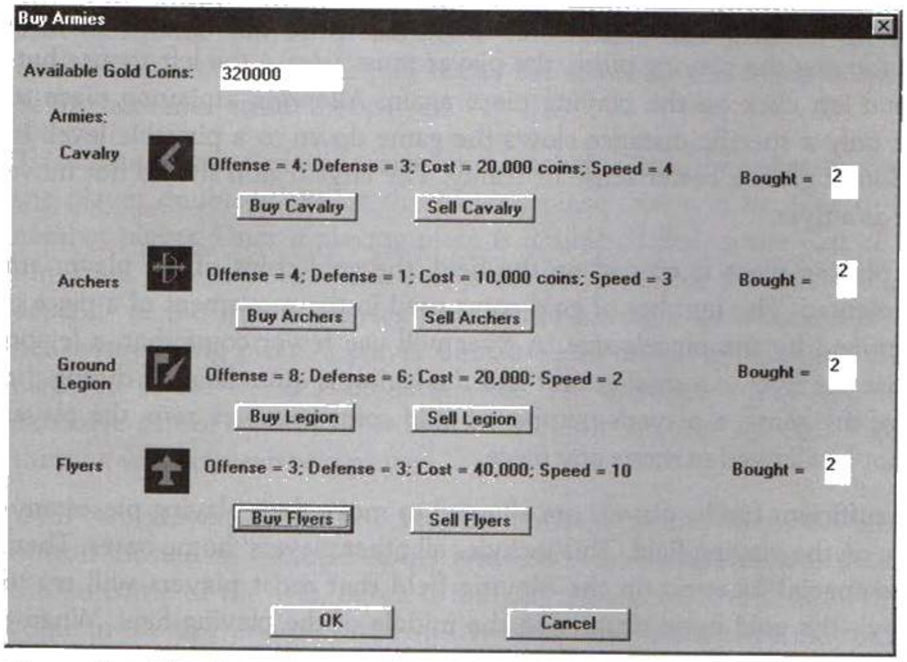
Figure 2.3 *The King's Reign operations window.*

playing pieces have been purchased and prepared for action, but are not necessarily ready to be used.

Placing Playing Pieces

To use one of the playing pieces, a player simply double-clicks on the appropriate playing piece in the list. When this occurs, a visual playing piece is added to the playing field and the words "Army Deployed" are put in the list of playing piece descriptions. Figure 2.4 shows the new situation.

As you can see, the playing piece has been put in the upper right corner of the playing field. This is the starting location for player 0. Each player starts in one of the four corners. Located in each corner is a home base. This home base is used to increase the defensive capabilities of a playing piece. We will discuss these capabilities shortly.


Buy Armies

Available Gold Coins: 320000

Armies:

Cavalry Offense = 4; Defense = 3; Cost = 20,000 coins; Speed = 4 Bought = 2

Archers Offense = 4; Defense = 1; Cost = 10,000 coins; Speed = 3 Bought = 2

Ground Legion Offense = 8; Defense = 6; Cost = 20,000; Speed = 2 Bought = 2

Flyers Offense = 3; Defense = 3; Cost = 40,000; Speed = 10 Bought = 2

Figure 2.4 *The King's Reign playing field and operations window when a playing piece is selected.*

With a playing piece on the field, players can do several things:

1. They can purchase more playing pieces, limited only by their gold coin reserves.
2. They can add more playing pieces to the field.
3. They can move their playing piece.

Moving Playing Pieces

Since we have already seen how to purchase and add playing pieces to the playing field, we will look at moving a playing piece. In order to move a playing piece, players move the mouse until the arrow is on the playing piece they want to move. The move is performed by pressing the left mouse button and holding it down. This action causes the playing piece to move wherever the mouse is moved. There is one catch, however. Each of the playing pieces has a specific movement value. This value causes the playing piece to move only a specific distance before it is stopped by the program. The mouse may be moving, but the playing piece is not. In order to continue moving the playing piece, the player must release the left mouse button and left click on the playing piece again. Allowing a playing piece to move only a specific distance slows the game down to a playable level. In addition, it gives a better sense of reality. The large legion should not move as far as a flyer.

As a playing piece is moved on the field, the gold coins of the player are decremented. The number of gold coins used in the movement of a piece is determined by the piece's size. A flyer will use fewer coins than a legion because the flyer is a smaller unit and the legion is quite large. If during the play of the game, a player's number of gold coins becomes zero, the player will not be allowed to move any piece.

With sufficient funds, players are allowed to move their playing pieces anywhere on the playing field. This includes all other players' home bases. There is one special location on the playing field that most players will try to occupy—the gold mine situated in the middle of the playing field. When a player puts a playing piece in the middle of the gold mine, the player's gold coins reserves will increase by 10 coins per second. However, if another player puts a playing piece in the gold mine as well, no players will receive gold. The two players will have to battle for possession of the gold mine.

Doing Battle

The main component of King's Reign is the battle. To win the game, you must battle until you destroy all of the other player's' playing pieces or until the other players give up. You must remember though, that you are unable to see another player's gold reserve. You may destroy all of the visible playing pieces, but a player could have more that you don't know about.

A battle or war is initiated by any player with a playing piece on the field. If you look back at Figure 2.3, you will see a small button with the label WAR. To the left of the button are the words "Not Okay". The complement to these words is "Okay For". A player having the "Okay For" words next to the WAR button can initiate a war. This player's machine has the war token. If the player does not use the token and initiate a war, it will automatically be sent to another player. If the words "Not Okay" are present, the player knows that either another player has the token or the token is currently being transferred to another player.

To initiate a war, a player waits for the token. Once the token is available, the player double-clicks on the playing piece that will be used to attack another player. Once a playing piece is double-clicked, some part of it will turn from gray to yellow. The part of the playing piece that turns to yellow depends on the specific playing piece. After having double-clicked on his or her own playing piece, a player double-clicks on the playing piece of the player he or she wants to do battle with. The opponent's playing piece will also have part of it turn yellow indicating that it has been selected. Figure 2.5 shows a war between two players.

With two playing pieces selected, the initiating player clicks on the WAR button located in the operations window. The computer will determine the outcome of the war, deselect the playing pieces, and display the outcome in the operations window. The computer uses a large matrix that is referenced based on the offensive value of the initiating player's playing piece and the defensive value of the opponent. These values reference a

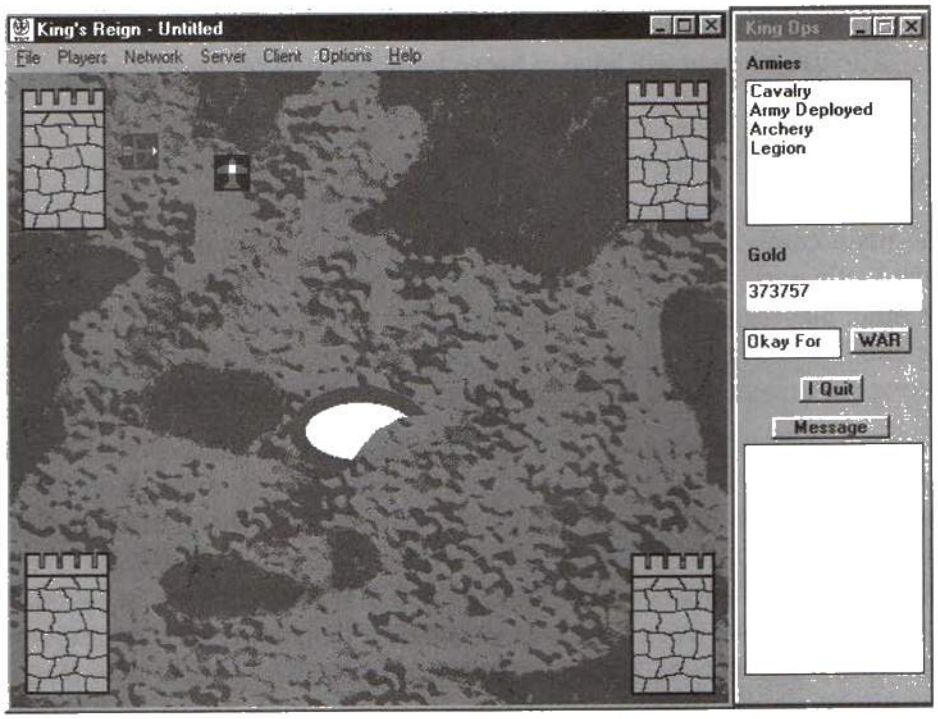


Figure 2.5 *A war between two players.*

location in a three-dimensional matrix. A random number generator is used to determine the outcome of the war based on the strengths of the playing pieces.

The outcomes available are:

- *NO DAMAGE:* neither playing piece is assessed points.
- *HALF DAMAGE:* the opponent in the war has both offense and defense points cut in half.
- *FULL DAMAGE:* the opponent in the war is destroyed.
- *ATTACKER DAMAGE:* the initiator is damaged by half.

Once the war has taken place, the current player no longer has access to the token until all other players have a chance to initiate a war.

Miscellaneous Operations

Looking back at Figure 2.3, you will see that the operations window has several other buttons and boxes in it. One of the buttons is labeled "I Quit". This is the surrender button. If a player clicks on this button, all of the player's pieces will be removed from all playing fields and the player's money will be zeroed out. This player will still be able to and must monitor the current game in progress. A player who surrenders is still technically in the game. If a player were to disconnect from the game, all of the other players would be unable to finish the game. The game in its present format does not have the ability to dynamically reconfigure itself upon the loss of a player. All of the communication data structures in all of the player's games are initialized to a specific number of players.

In many cases, the players in the game will be using phone lines to connect to each other. In this case, there should be a way for the players to communicate with each other. This is the purpose of the Message button and the associated box under the button. By clicking on the Message button, any player can communicate with either one other player or all others.

Game Summary

Depending on the players, a game of King's Reign can last 5 minutes or many hours. Careful purchasing of equipment and a little luck with the random number generator can be a winning combination. Now that you have some idea about how a game is played and the different components involved, we can look at the design of the game.

King's Reign Design

The main idea behind the code design of King's Reign is to take full advantage of the Microsoft Foundation Classes (MFCs) available in the Visual C++ package. MFC provides the Windows programmer with a vast number of object classes, which significantly simplifies programming a Windows application. These object classes allow the entire game to be created in under 4,000 lines of code. This includes all of the code for dialog boxes and resources, as well as two different types of communication protocols.

Another concern for the design of the game was to provide a user-friendly interface. Although a graphical user interface such as that used in Microsoft Windows itself is designed to be user friendly, a programmer can easily override this friendliness by making the setup of an application difficult. We try to avoid this by doing most all of the setup behind the scenes. Only a few details must be provided by the user.

The game consists of a collection of objects. The main object is called CNetwar2View. This object creates the windows and interface between the computer and the user. As the game is played, various other objects are created. Each of the dialog boxes presented to the user is an object. For instance, when the user wants to purchase playing pieces, the BUY_EQUIPMENT object is invoked. This object is a CDialog object which means it is used to display a dialog box.

When a player determines whether they are the server or a client, an appropriate object is created to hold information about this player or in the case of a client, an object is created describing the server. These objects will interact with the CNetwar2View object when information about the user is needed.

Below you will find a description of each of the objects used in the game. They all have a specialized function which they perform.

BUY_EQUIPMENT

The BUY_EQUIPMENT object is a class derived from the CDialog class of the MFC. The object definition is shown in Listing 2.1.

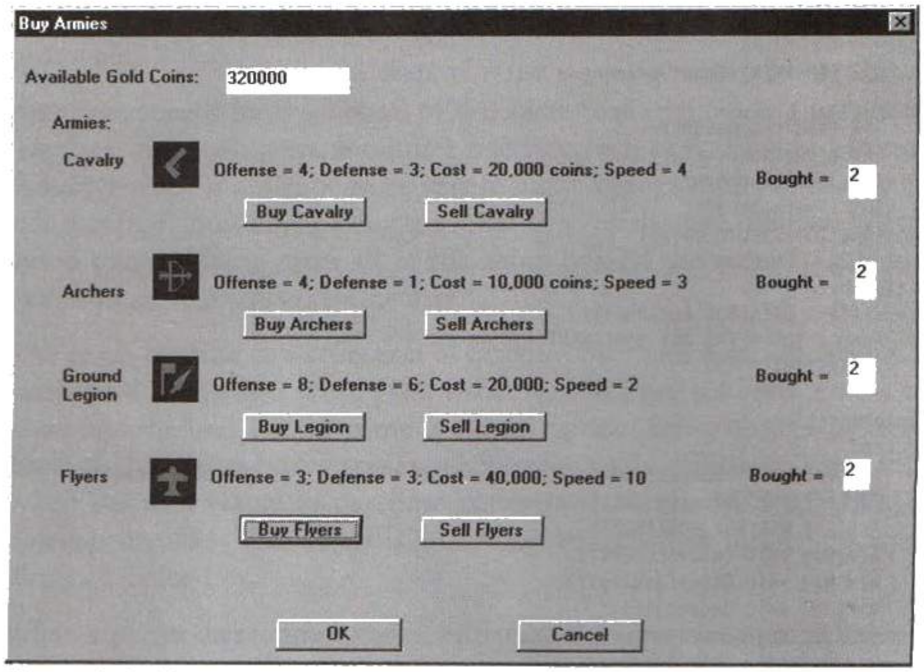
Listing 2.1: The BUY_EQUIPMENT Object Definition

```
// buy_equi.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// BUY_EQUIPMENT dialog

class BUY_EQUIPMENT : public CDialog
{
```

```
// Construction
public:
        BUY_EQUIPMENT(CWnd* pParent = NULL); // standard constructor

        int cavalrys_bought;
        int archers_bought;
        int legions_bought;
        int flyers_bought;
        long total_gold_coins;

// Dialog Data
        //{{AFX_DATA(BUY_EQUIPMENT)
        enum { IDD = IDD_BUY_EQUIPMENT };
        //}}AFX_DATA

// Implementation
protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        // Generated message map functions
        //{{AFX_MSG(BUY_EQUIPMENT)
        afx_msg void OnBuyArchers();
        afx_msg void OnBuyCavalrys();
        virtual void OnCancel();
        virtual void OnOK();
        afx_msg void OnSellArchers();
        afx_msg void OnSellCavalrys();
        afx_msg void OnBuyFlyers();
        afx_msg void OnBuyLegion();
        afx_msg void OnSellFlyers();
        afx_msg void OnSellLegion();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

The dialog box created from the BUY_EQUIPMENT object is shown in Figure 2.6. It is a standard dialog window that includes 4 icons, 10 pushbuttons, 5 edit input lines, and static text. This object is generated by a click on the view object's menu. Once created, all of the functionality is handled in the buy_equi.cpp code file.

The object allows for the user to click on any of the 10 pushbuttons. Four of the pushbuttons correspond to a purchase of equipment. Looking at Listing 2.1, we can see that there are four functions called OnBuy*. Each of the functions relates to a specific playing piece. A click on a Buy * pushbutton will execute the appropriate function. The functions decrement the total amount of gold coins in reserve and increment the number of playing pieces purchased.



Available Gold Coins: 320000

Armies:

Cavalry Offense = 4; Defense = 3; Cost = 20,000 coins; Speed = 4 Bought = 2

Archers Offense = 4; Defense = 1; Cost = 10,000 coins; Speed = 3 Bought = 2

Ground Legion Offense = 8; Defense = 6; Cost = 20,000; Speed = 2 Bought = 2

Flyers Offense = 3; Defense = 3; Cost = 40,000; Speed = 10 Bought = 2

Figure 2.6 *The BUY_EQUIPMENT object dialog box.*

The functions must also have a degree of security. This comes in the form of monitoring the total number of gold coins available. If a player tries to purchase a playing piece but does not have enough gold coins, the function will let the player know and the purchase will not go through. The number of playing pieces purchased are kept in four public variables in the object. These variables are called *_bought, where * stands for the four different playing piece names.

Four of the remaining pushbuttons are Sell * pushbuttons. These pushbuttons relate to the four OnSell* functions defined in the object. These functions increment the total number of gold coins for the player and decrement the total number of playing pieces purchased. Again, the function must monitor the transactions that are occurring. Instead of monitoring only the total number of gold coins, the function also must not allow the user to sell back a playing piece that he or she has not purchased. Thus, if the corresponding playing piece variable for this function is 0, the function will let the player know that he or she doesn't have any playing pieces of this kind available to sell back.

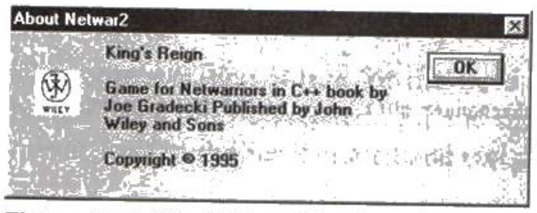
The last two pushbuttons are OK and Cancel. With a click on the OK pushbutton, the window is destroyed and the number of playing pieces purchased is retrieved by the view object. The view object is able to access the playing piece variables of the BUY_EQUIPMENT object because they were declared as PUBLIC. Check back to Listing 2.1 to verify this. A click of the Cancel pushbutton causes all of the playing piece counts to be zero, and the view object will not register any purchased playing pieces.

The About Dialog Box (CAboutDlg)

The CAboutDlg is a dialog box for the About menu item of the view object. It displays a dialog box with a single OK pushbutton in it as well as several lines of text. There is also the application's icon in the dialog box. The only option available to the user is to read the dialog text and click on the OK button. The CAboutDlg dialog box is shown in Figure 2.7.

The Modem Dialog Box (COMMINPUT)

The COMMINPUT dialog box is a class derived from the CDialog class in the MFC. Its purpose is to retrieve information from users about their modems. This dialog window is created by the view object and is only created when the players are using the modem to direct play and not when they are using the modem for PPP Internet play. The dialog box created by the COMMINPUT object is shown in Figure 2.8.



King's Reign

Game for Netwarriors in C++ book by Joe Gradecki Published by John Wiley and Sons

Copyright © 1995

Figure 2.7 *The CAboutDlg object dialog box.*

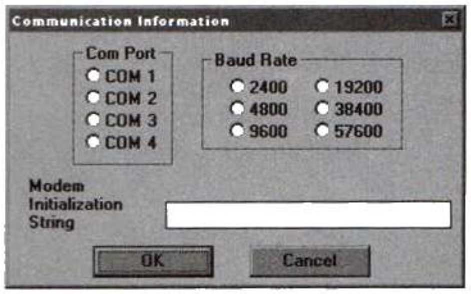


Figure 2.8 *The COMMINPUT object dialog box.*

The object definition is shown in Listing 2.2.

Listing 2.2: The COMMINPUT Object Definition

```
// omminput.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// COMMINPUT dialog

class COMMINPUT : public CDialog
{
public:
  int current_baud,
      old_baud,
      current_com,
      old_com;

// Construction
public:
      COMMINPUT(CWnd* pParent = NULL); // standard constructor

// Dialog Data
      //{{AFX_DATA(COMMINPUT)
      enum { IDD = IDD_COMM_INPUT };
      CString m_initialization_string;
      //}}AFX_DATA
// Implementation

protected:
      virtual void DoDataExchange(CDataExchange* pDX);      // DDX/DDV support

      // Generated message map functions
      //{{AFX_MSG(COMMINPUT)
      afx_msg void OnBaud19200();
      afx_msg void OnBaud2400();
```

```
        afx_msg void OnBaud38400();
        afx_msg void OnBaud4800();
        afx_msg void OnBaud57600();
        afx_msg void OnBaud9600();
        afx_msg void OnCom1();
        afx_msg void OnCom2();
        afx_msg void OnCom3();
        afx_msg void OnCom4();
        virtual void OnOK();
        virtual void OnCancel();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

        void turn_off_baud(int current_baud);
        void turn_off_com(int com);
};
```

The dialog box includes a group of four radio buttons labeled COM Port, eight radio buttons labeled Baud Rate, an edit line for a modem initialization string, and an OK button. The controls are handled by the source code file comminpu.cpp.

When this dialog box appears, the user must click on one of the radio buttons in the COM port group and one of the buttons in the Baud Rate group. The radio buttons are themselves MFC objects, as we will see in Chapter 4. What this means to us is that we don't have to do much programming to control the buttons. If a user clicks on the COM 1 radio button, we don't have to tell the control to put a selection circle in the radio button. In addition, if the user clicks on COM 2 after clicking on COM 1, we don't have to change the selection circle. The radio button object will handle all of this itself. We will see more about this in a later chapter. The only thing that we have to do when a radio button is clicked is set a PUBLIC variable called *current_com* to the appropriate COM port number. Thus looking at Listing 2.2, you will see the COM variable defined as well as four functions called OnCom1, OnCom2, OnCom3, and OnCom4. When a COM radio button is clicked, the appropriate function is called.

The same situation is true for the Baud Rate group of radio buttons. Each button has a function related to it and the functions set the PUBLIC variable *current_baud* to the appropriate baud rate value.

After the users have selected their modem's COM Port and Baud Rate, they are able but not required to enter a modem initialization string in the edit line provided. Once users are satisfied with their selections, they must click on the OK button. The view object will extract the appropriate information from the dialog box.

The Input Name Dialog Box (INPUTNAME)

The INPUTNAME object is an object derived from the CDialog class in the MFC. The object listing is shown in Listing 2.3.

Listing 2.3: The INPUTNAME Object Definition

```
// inputnam.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// INPUTNAME dialog

class INPUTNAME : public CDialog
{
// Construction
public:
       char name[30];
       int type;
       char total_players[5];

       INPUTNAME(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
       //{{AFX_DATA(INPUTNAME)
       enum { IDD = IDD_INPUT_NAME };
       CString        m_input_name;
       int            m_total_players;
       //}}AFX_DATA

// Implementation
protected:
       virtual void DoDataExchange(CDataExchange* pDX);     // DDX/DDV support

       // Generated message map functions
       //{{AFX_MSG(INPUTNAME)
       afx_msg void OnClickedOk();
       afx_msg void OnOK();
       afx_msg void OnRadioClient();
       afx_msg void OnRadioServer();
       //}}AFX_MSG
       DECLARE_MESSAGE_MAP()
};
```

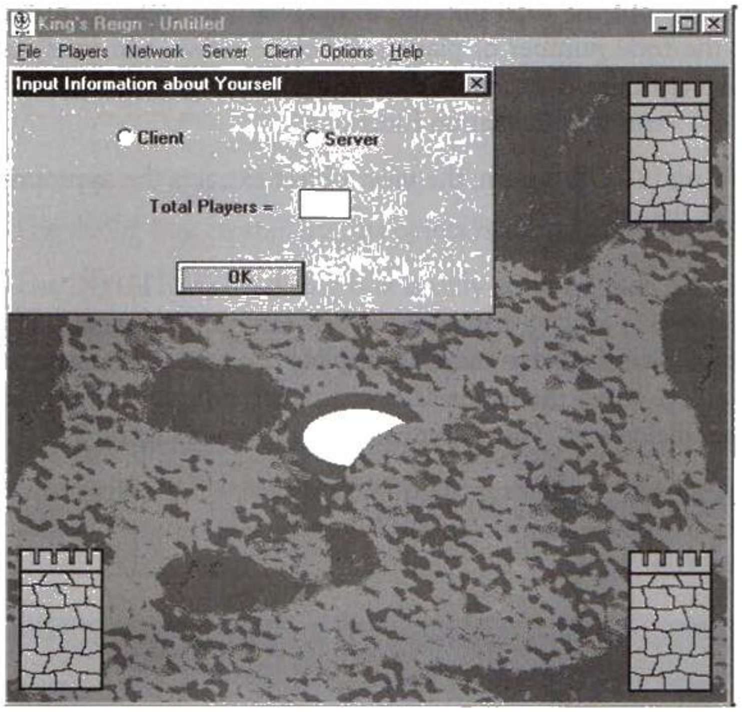

King's Reign - Untitled
File Players Network Server Client Options Help
Input Information about Yourself
Client
Server
Total Players =
OK

Figure 2.9 *The INPUTNAME object dialog box.*

The dialog box that appears as a result of the INPUTNAME object in shown in Figure 2.9.

The purpose of the INPUTNAME object is to retrieve information about you as a player. The dialog box includes a space for your name, two radio buttons, an edit line for the total number of players, and an OK button. This object is called by the view object when the user clicks on the Input Players menu item. When the dialog box appears, users will typically enter their names in the edit lines provided, although this is not necessary for game play.

Next, the player will select whether he or she is a client or a server. The difference is that a client will "call" the server and the server will "receive" calls from the clients. When a radio box is clicked, the system will call either the OnClient or the OnServer function defined in the INPUTNAME object. These functions set a type PUBLIC variable to either 0 or 1 to indicate whether the player is a client or a server.

Finally, the user must input the total number of players. The system checks to make sure that the total number of players is between two and four. As we will see, this is very important, since the application software uses this number of players for various initialization functions.

Once the user clicks on the OK button, the view object extracts the appropriate information from the dialog box.

INPUTPLAYER

The INPUTPLAYER object is a derived CDialog MFC class. This object is used by the view object when the user selects the client radio button on the INPUTNAME object dialog box. The object listing is shown in Listing 2.4.

Listing 2.4: The INPUTPLAYER Object Definition

```
// inputpla.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// INPUTPLAYER dialog

class INPUTPLAYER : public CDialog
{
// Construction
public:
    char address[20];
    char name[30];
    int type;      // 0 = client, 1 = server

public:
       INPUTPLAYER(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
       //{{AFX_DATA(INPUTPLAYER)
       enum { IDD = IDD_INPUT_PLAYER };
       CString m_input_player_internet;
       CString m_input_player_name;
       //}}AFX_DATA

// Implementation
protected:
       virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

       // Generated message map functions
```

```
        //{{AFX_MSG(INPUTPLAYER)
        virtual void OnCancel();
        virtual void OnOK();
        afx_msg void OnClickedOK();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

The dialog box created by this object is shown in Figure 2.10.

The INPUTPLAYER dialog box is only shown when the user is a client. Since all clients must register with a server, the system needs to obtain server information from the user. This dialog box has an edit line for either the telephone number or the Internet address of the server.

The edit line for the telephone or Internet address must be filled in. The information put into this edit line is available to the view object in the variable *m_input_player_internet*. This information will be used to contact the server.

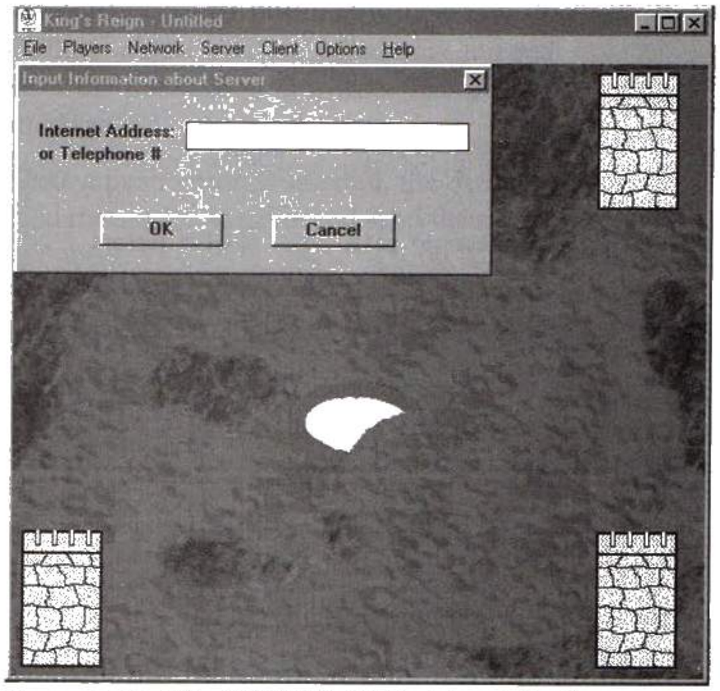

King's Reign - Untitled
File Players Network Server Client Options Help
Input Information about Server
Internet Address:
or Telephone #
OK
Cancel

Figure 2.10 *The INPUTPLAYER object dialog box.*

The MESSAGE Object

The MESSAGE object is derived from the CDialog class. The object definition is shown in Listing 2.5.

Listing 2.5: The MESSAGE Object Definition

```
// message.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// MESSAGE dialog

class MESSAGE : public CDialog
{
public:
   int sendto;

// Construction
public:
       MESSAGE(CWnd* pParent = NULL); // standard constructor

// Dialog Data
       //{{AFX_DATA(MESSAGE)
       enum { IDD = IDD_MESSAGE_INPUT };
       CString m_message_input;
       //}}AFX_DATA

// Implementation
protected:
       virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

       // Generated message map functions
       //{{AFX_MSG(MESSAGE)
       afx_msg void OnSendMessage1();
       afx_msg void OnSendMessage2();
       afx_msg void OnSendMessage3();
       afx_msg void OnSendMessage4();
       afx_msg void OnSendMessageAll();
       virtual void OnCancel();
       //}}AFX_MSG
       DECLARE_MESSAGE_MAP()
};
```

The dialog box created by this object is shown in Figure 2.11.

As we saw in the description of the game earlier, there are times when the players may want to communicate with each other. They perform the communication using the MESSAGE object dialog box. This dialog box is acti-

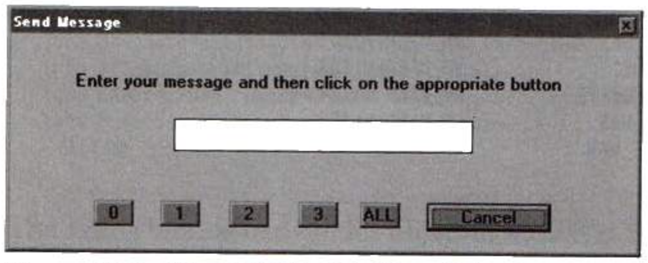


Figure 2.11 *The MESSAGE object dialog box.*

vated when the player clicks on the Message pushbutton in the Operations window.

The MESSAGE dialog includes a single edit line for the message and six pushbuttons. Obviously, users should enter their message in the edit line first. The message is put into the variable *m_message_input*. The user will then click on one of the pushbuttons to indicate which player should receive the message. The object includes functions to handle each of the pushbuttons. When a particular pushbutton is clicked, the corresponding function sets a PUBLIC object variable called *send_to* equal to either the player number to send the message to or a value indicating that all players should receive the message.

Once a pushbutton is selected, the dialog box is closed. The view object will read the *send_to* variable and send the message to the appropriate player(s).

The Operations Dialog Box (OPERATIONSDIALOG)

The OPERATIONSDIALOG is another CDialog-derived class but differs from those above in that it is a modeless dialog box. It will remain on the screen throughout the game. The object definition is given in Listing 2.6.

Listing 2.6: The OPERATIONSDIALOG Object Definition

```
// operatio.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// OPERATIONSDIALOG dialog

#define WM_OPERATIONS_ADD_ARMIES                WM_USER + 100
```

```
#define WM_OPERATIONS_UPDATE_GOLD          WM_USER + 101
#define WM_OPERATIONS_UPDATE_MESSAGE       WM_USER + 102
#define WM_OPERATIONS_NEW_GAME             WM_USER + 103
#define WM_OPERATIONS_PLAYER_QUITS         WM_USER + 104
#define WM_OPERATIONS_ENABLE_WAR           WM_USER + 105
#define WM_OPERATIONS_DISABLE_WAR          WM_USER + 106

typedef struct _message_list
{
   char message[50];
   struct _message_list *next;
} MESSAGE_LIST;

class OPERATIONSDIALOG : public CDialog
{
public:
     CView* m_pView;     //whose view created us
     int total_armies;
     int list_box[50];
     char entries[50][20];
     int selected_army;
     long total_gold;
     int total_messages;
     char new_message[50];
     MESSAGE_LIST *messages;

// Construction
public:
     OPERATIONSDIALOG(CWnd* parent = NULL);
      OPERATIONSDIALOG(CView* pView);  // standard constructor
      BOOL Create();

// Dialog Data
      //{{AFX_DATA(OPERATIONSDIALOG)
      enum { IDD = IDD_OPERATIONS_DIALOG };
      CString   m_operations_available_gold;
      CString   m_war_statement;
      //}}AFX_DATA

// Implementation
protected:
      virtual void DoDataExchange(CDataExchange* pDX);      // DDX/DDV support

      // Generated message map functions
      //{{AFX_MSG(OPERATIONSDIALOG)
      afx_msg void OnWarButton();
      afx_msg void OnDblclkAvailableArmies();
      afx_msg void OnOperationsIQuit();
      afx_msg void OnOperationsMessageButton();
      //}}AFX_MSG
   long OnAddArmies(UINT wParam, LONG lParam);
```

```
    long OnUpdateGold(UINT wParam, LONG lParam);
    long OnUpdateMessage(UINT wParam, LONG lParam);
    long OnNewGame(UINT wParam, LONG lParam);
    long OnWarDo(UINT wParam, LONG lParam);
    long OnWarUnDo(UINT wParam, LONG lParam);
     DECLARE_MESSAGE_MAP()
};
```

The dialog box created by this object in shown in Figure 2.12.

Because the main view object holds a war in its entire client area, a second window is created to hold the operational details of the game. As you can see from Listing 2.5 and Figure 2.12, the dialog box includes a list box for the purchased playing pieces, an edit box for the player's gold coin count, an edit box for the war token, a war button, a surrender button, a message button, and a list box for system and player messages.

The view object that controls the game communicates with the operations window using a series of messages and PUBLIC variables. The first and most common message deals with the edit box for the player's gold coin count. When users input their information about the players in the game, the view object will

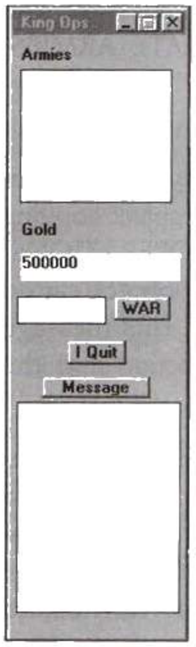


Figure 2.12 *The OPERATIONSDIALOG object dialog box.*

set the OPERATIONSDIALOG PUBLIC variable *m_gold_coins* equal to a starting value. After this, a message is posted to the operations window object indicating that it should display the value in *m_gold_coins* in the edit box. The message used is WM_OPERATIONS_UPDATE_GOLD. When this message is received by the OPERATIONSDIALOG, the function ON_UPDATE_GOLD is called and the value is displayed in the correct location.

When a player purchases playing pieces, the names of the pieces must appear in the operations window in order for the pieces to be selected. The OPERATIONSDIALOG object has the variables *total_armies, list_box[50],* and *entries[50][20]* defined. These variables are used to hold the information about the armies purchased. After a playing piece purchase, the view object will put an index to the object in the variable *list_box* at the *total_armies* location. This is the first unoccupied array location. In addition, the name of the playing piece will be placed in the variable *entries,* also in the *total_armies* location. Once inserted, the *total_armies* variable is incremented. After all of the playing pieces purchased are in their appropriate array locations, the message WM_OPERATIONS_UPDATE_ARMIES is sent to the OPERATIONSDIALOG object.

Upon receipt of the WM_OPERATIONS_UPDATE_ARMIES message, the OPERATIONSDIALOG object will call the function ON_UPDATE_ARMIES. This function will fill the list box on the operations window with the purchased armies.

The second list box in the window is for messages from the system or from other players. A message is inserted into the list by the view object copying the text into the PUBLIC variable *new_message* located in the OPERATIONSDIALOG object. A message is then sent from the view object called WM_OPERATIONS_UPDATE_MESSAGE. When received, this message causes the function OnUpdateMessage to be called. This function puts the new message in the message list box.

The remaining controls in the dialog box are pushbuttons. Each of the pushbuttons operates in the same way. When the user clicks on a pushbutton, a function associated with the pushbutton is called. Since we are in the operations window, we must let the view object know (because it is the object in control of the game). This is done by sending a message to the view object.

The war button uses the message WM_WAR_BUTTON, the message button uses the message WM_MESSAGE_BUTTON, and the surrender button uses the message WM_QUIT_BUTTON. Each of these messages is registered with the view object and is handled accordingly.

WAITCLIENT

WAITCLIENT is a derived CDialog class. Its purpose is to display simple messages to the user. The object is invoked by the view object at several different places.

CSprite

The CSprite class is a class designed to handle bitmaps on the screen. It is adapted from a sprite class described in an article by Michael J. Young in the June 1993 issue of *BYTE Magazine.* The class definition is shown in Listing 2.7.

Listing 2.7: The CSprite Class Definition

```
/////////////////////////////////////////////////////////////////////////////
//                                                                       //
// SPRITE.H: Header file for the CSprite class.                          //
//                                                                       //
/////////////////////////////////////////////////////////////////////////////
#include <time.h>

class CSprite
{
public:
   int XOffset, YOffset,
       mHeight, mWidth,
       mX, mY;
   BOOL active;           // our playing piece could be instantiated but not active
   int offense;
   int defense;
   int player;               // the owner of this playing piece
   BOOL show;             // If TRUE then we have shown this sprite
   char type[10];         // the type of playing piece
   int units_per_move:  // speed of mouse movement for piece
   int startx, starty:
   BOOL war;
   int setby;
private:
   CBitmap* mHImage;
   CBitmap* mHMask;
   CBitmap* mHSave;
   CBitmap* mHHighlight;
   CBitmap* mHHighlightmask;
   CBitmap* mtemp;
   CBitmap* mtempmask;

   BOOL switched;

public:
   CSprite ();
   ~CSprite ();

   void GetCoord (CRect *Rect);
   BOOL Hide (CDC* HDc);
   BOOL Initialize (CDC* Hdc,
                              CBitmap* HMask,
                              CBitmap* HImage,
                              CBitmap* HHighlight,
                              CBitmap* HHighlightmask,
                              BOOL Active,
                              int Offense,
                              int Defense,
                              int Player,
                              char *Type,
                              int Speed
                              );
  BOOL MoveTo (CDC* HDc, int X, int Y);
  BOOL Redraw (CDC* HDc);
  BOOL Start (CDC* HDc, int X, int Y);
  BOOL ReplaceBitmap();

};
```

All of the playing pieces in the game are defined as CSprite objects. This allows each playing piece to be responsible for its own characteristics such as offense and defense points. The sprite objects are controlled by the view object and defined in the two-dimensional array playing_pieces.

A sprite is initialized by the class method Initialize(). This function has the task of recording the different bitmaps that make up the playing piece. Every piece is different. It is either a flyer, archer, legion, or cavalry piece and is a different color for each player. This dictates that each sprite be initialized individually. The object will copy the appropriate bitmaps into internal PRIVATE variables.

Each time the view object is redrawn on the screen, all of the playing piece objects must be redrawn as well. When the playing pieces are first placed on the playing field, they must be drawn with the Start() class method. This function places the sprite at a specific location on the screen. Each subsequent draw will use the method Redraw(). This function uses the internal coordinates of the sprite to redraw it.

Probably the most important method in the CSprite class is MoveTo(). This function allows the sprite to be moved to a different location on the screen. As we will see in Chapter 5, this function must handle all of the screen updates correctly in order for the background of the playing field to remain solid. The method takes advantage of several internal data structures to keep track of the playing field background.

In the description of the game earlier, we saw that a playing piece changed when it was selected for a war. The change came about by coloring part of the playing piece bitmap yellow. In order to accomplish this, a method called ReplaceBitmap() is used. This function replaces the normal unselected bitmap of the sprite with the yellow bitmap. The function works as a toggle in that when it is first called, the normal bitmap is replaced with the yellow one. When the function is called a second time, the yellowed bitmap is replaced with the normal bitmap.

CPlayers

The CPlayers class is another class specifically designed to handle the players in the game. The class definition is shown in Listing 2.8.

Listing 2.8: The CPlayers Class Definition

```
#include "winsock.h"

class CPlayers
{
public:
   int socket;
   struct sockaddr_in connection;
   int type;         // 0 = client, 1 = server
   long gold_coins;
   BOOL ready_for_game;
   int kills;
private:
   char name[20];
   char address[15];
   int active;
public:
   CPlayers();
   ~CPlayers();

 void set_address(char *Address);
 void get_address(char *Address);

 void set_name(char *Name);
 void get_name(char *Name);

 void set_active(int Active);
 int get_active();
};
```

The main purpose of the CPlayers class is to keep track of specific information about each player. When the user was prompted with the INPUTPLAYER class dialog box, the information he or she gave was entered into an object of this class. The name and Internet address or telephone number of the server player is kept in two PRIVATE variables, *name* and *address*. These variables are only used if the variable *type* is a 1, indicating this player is a server.

Because we are designing a multiplayer game, we know that there must be communication channels between the players. One task of the CPlayers class is to keep track of the specific channels for the players using the Internet. The variable *connection* is used for this purpose.

The CPlayers class also include variables for the total number of gold coins available to this player, a variable used to count the number of kills registered by the player during wars, and a Boolean variable used to indicate the player is ready for the game to start.

CBattle

The CBattle class is a specialized class for handling the outcome of wars in the game. The definition of the class is given in Listing 2.9.

Listing 2.9: The CBattle Class Definition

```
#define NO_DAMAGE 0
```

```
#define HALF_DAMAGE 1
#define BOTH_DAMAGE 2
#define ATTACKER_DAMAGE 3
#define DESTROYED 4


class CBattle
{
 public:
    get_battle_result(int offense, int defense);

 private:
   int battle_matrix[13][13][6];

 public:
   CBattle();
   ~CBattle();
};
```

The CBattle class is fairly simple. It includes two methods; one for initialization and one for reading the war matrix. The war matrix is a 13 × 13 × 6 three-dimensional matrix. Each of the column entries corresponds to an offense point, and each of the row entries corresponds to a defense point.

When a war takes place, the view object makes a call to get_battle_result(), giving the offense and defense points of the playing pieces. The function moves to the location specified by the offense and defense value. A random number between 0 and 5 is chosen. The function extracts the war outcome value relating to the three-dimensional location (offense, defense, random_number). This is returned to the caller.

CServer

The last class is called CServer. It is used by the view object of the player that is designated as the server in the game. The class definition is shown in Listing 2.10.

Listing 2.10: The CServer Class Definition

```
#include "winsock.h"

class CServer
{
 public:
    int socket;
    struct sockaddr_in connection;
    BOOL active_connections[3];
    int server_player_number;      // player that is the server in our list
    int total_connected;           // total players connected to the server
    int clients[3];                          // sockets for connected players
    struct sockaddr_in clients_addr[3]; //structures for connected players

  public:
    CServer();
    ~CServer();
};
```

An object of CServer is only initiated when the game is being played over the Internet. As we will see in the next section, a single server is used to service all of the players in the game. This server has the responsibility to farm messages to each of the players in the game. To do this, it will have from one to three connections active, depending on the number of players in the game. The variables in the CServer class keep track of the connections and other housekeeping tasks.

King's Reign View Object Messages

With all of the objects defined, we have to have a central object that will control the application. Throughout the description of the objects, the object mentioned for this control is the view object. The view object is the window the user sees on the screen. It has the main message loop of the application. As you may know, a Windows application is not a traditional top-down execution type. It is event driven, responding to messages sent from the Windows system as well as the application itself. Messages come from four main areas: menu, communications, mouse, and operations. To show how the view object coordinates the objects listed above, we will go through each of the messages that our application will receive and respond to.

Menu Commands

Almost every Windows application has a menu. If you look back at Figure 2.1, you will see the menu at the top of the view object. Some of the menu options are popups (a secondary menu appears); others are just single options.

ID_FILE_NEW

The ID_FILE_NEW message is sent to the view object when the player clicks on the New Game menu option. The view object must reset itself as well as let the other players know that a new game is about to be played.

ID_PLAYERS_INPUTPLAYERS

The ID_PLAYERS_INPUTPLAYERS message is sent to the view object when the player clicks on the Input Players menu item. The view object creates an INPUTNAME object to receive information about the player. After the player clicks on the OK button of the INPUTNAME object, the view object creates *X* number of CPlayers objects, where *X* is the number of players registered in the INPUTNAME object. Depending on whether or not the player is a client or a server, the view object will create an INPUTPLAYER object to receive information about the server. If the current player is the server for the game, the view object will create a CServer object to hold the server information.

ID_PLAYERS_BUYEQUIPMENT

The ID_PLAYERS_BUYEQUIPMENT message is sent to the view object when the player clicks on the Buy Equipment menu item. The view object creates a BUYEQUIPMENT object. Once the player enters the appropriate information into the BUYEQUIPMENT object, the view object processes the information. Depending on the playing pieces purchased, the view object will create the appropriate number of CSprite objects to handle all of the purchased equipment.

ID_NETWORK_MODEM

The ID_NETWORK_MODEM message is sent to the view object when the player clicks on the Modem menu item. The view object creates a COMMINPUT object to gather information about the user's modem.

ID_NETWORK_TCP

The ID_NETWORK_TCP message is sent to the view object when the player clicks on the TCP/IP menu item. The view object does not create any objects to handle Internet communications. However, it does initialize the Windows protocol for Internet communications.

ID_CONNECT

The ID_CONNECT message is sent to the view object when the player clicks on the Client menu item. The view object attempts to connect with the server based on the information the user entered in the INPUTPLAYER object.

ID_WAITFORCONNECT

The ID_WAITFORCONNECT message is sent to the view object when the player clicks on the Server menu item. The view object will monitor the number of players in the game and wait for the appropriate connections to be established.

ID_OPTIONS_SOUNDS

The ID_OPTIONS_SOUND message is sent to the view object when the player clicks on the Options menu popup and then clicks on Sounds. The view object simply sets a toggle variable.

ID_OPTIONS_ACTIVATEACKNOWLEDGEMENTS

The ID_OPTIONS_ACTIVATEACKNOWLEDGEMENTS message is sent to the view object when the player clicks on the Options menu popup and then clicks on Activate Acknowledgements. The view object sets a toggle variable.

ID_OPTIONS_DOEXTRAINVALIDATE

The ID_OPTIONS_DOEXTRAINVALIDATE message is sent to the view object when the player clicks on the Options menu popup and then clicks on Extra Invalidate. The view object sets a toggle variable.

Communication Commands

Although game communication will be discussed in the next section, we need to mention the messages that the view object receives. This version of Netwarriors in C++ allows two different types of communication channels: modem and Internet. Each of the types requires a number of different messages.

Serial Port

Because only two players are available in a modem game, the application only receives information from one channel. Once established, this channel is able to communicate with the view object using the WM_COMMNOTIFY message. When information comes available on the modem line, the view object receives this message. The view object must retrieve the information as soon as possible and act upon that information.

Internet

An Internet game is able to handle up to four players. Each of the players communicates on different channels with the server. We will see more about this in the next section. The server requires six different messages and a client only one.

The server messages are: MSG_FROM_PLAYER1, MSG_FROM_PLAYER2, MSG_FROM_PLAYER3, MSG_ACCEPT, MSG_ACCEPT1, and MSG_ACCEPT2. The latter three messages are used when players are initially establishing a channel with the server. The three former messages are used once the channels are established. When the view object of the server receives one of these three messages, it knows information has come from a player and it must retrieve and act upon the information.

The client message is SERVER_MSG. When the view object of a client machine receives this message, the view object must retrieve the information on the channel and act upon it.

Mouse Commands

The user interface with the game is entirely through the mouse. The majority of the mouse movements deal with playing piece movement. There are five messages the view object responds to.

WM_LBUTTONDOWN

The WM_LBUTTONDOWN message is sent to the view object when the user presses the left mouse button. If the mouse cursor is in the client area of the application, not the menu, the view object will determine if the cursor is on one of the player's playing pieces. If this is the case, the view object will prepare the playing piece to be moved. The system will be instructed to send WM_MOUSEMOVE messages to the view object.

WM_LBUTTONUP

The WM_LBUTTONUP message is sent to the view object when the user releases the left mouse button. If a playing piece had previously been set up to be moved, the movement of the piece is terminated. The final position of the playing piece is sent to the other players in the game. At this point, we don't want the system to send the WM_MOUSEMOVE messages since we are no longer moving a playing piece. The system will automatically stop sending the WM_MOUSEMOVE message to our window.

WM_MOUSEMOVE

The WM_MOUSEMOVE message is sent to the view object when the user moves the mouse button and the system has requested the movement message be sent to the view object. Upon each message, the playing piece selected by the WM_LBUTTONDOWN message is moved to the new position of the mouse.

WM_LBUTTONBLCLK

The WM_LBUTTONBLCLK message is sent to the view object when the user double-clicks the mouse. If the mouse cursor is on one of the player's playing pieces and the player currently has the war token, the playing piece will be highlighted and a war will be allowed to take place.

WM_RBUTTONBLCLK

The WM_RBUTTONBLCLK message is sent to the view object when the user double-clicks the mouse's right button. If the mouse cursor is on one of the player's playing piece, the current statistics of the playing piece will be displayed to the user.

Operations Commands

The *operations* dialog box object has several buttons that the view object must respond to. In addition, that application uses a couple of timers.

WM_TIMER

The WM_TIMER message is sent to the view object when several different timers expire. The view object must respond to the timers.

WM_ARMY_SELECTED

The WM_ARMY_SELECTED message is sent to the view object when the user double-clicks on any of the playing pieces listed in the operations window Armies list box. The view object must place the playing piece on the screen.

WM_WAR_BUTTON

The WM_WAR_BUTTON message is sent to the view object when the user clicks on the War button in the operations window. The view object must initiate and process the current war, if any.

WM_QUIT_BUTTON

The WM_QUIT_BUTTON message is sent to the view object when the user clicks on the I Quit button in the operations window. The view object must terminate the current player.

WM_MESSAGE_BUTTON

The WM_MESSAGE_BUTTON message is sent to the view object when the user clicks on the Message button. The view object creates a MESSAGE object and processes a message to the other players.

King's Reign Communications

The cornerstone of the King's Reign game is the ability to allow up to four players to interact in a single game using the Internet and Microsoft Windows. The game can also be played by two players using the modem. In this section, we will briefly look at the design strategy of the communication protocols.

Internet

The Internet communication ability of our game has to be weighed carefully with performance and ease of programming. There are two ways in which we could accomplish the communications. The first is through a total communication scheme. Figure 2.13 shows the necessary channels involved.

As you can see from Figure 2.13, each player in our game would be required to have three active connections. The application would not only have to send information along each channel but would also have to monitor each channel. Whether or not this is overdoing the communications for the game is debatable, but a simpler system is in order. Figure 2.14 shows the scheme used in our game.

The Client/Server protocol is widely popular in the distributed computer field. In this protocol there is a central computer called the server which operations clients are able to use.

Clients register with the server at the start of our game. The server has three open channels where the clients are able to register. If there are less than four players in the game not all of the channels will be used. All of the clients have a single channel open to the server. During the execution of the game, the server can communicate with the clients through a single channel. In this

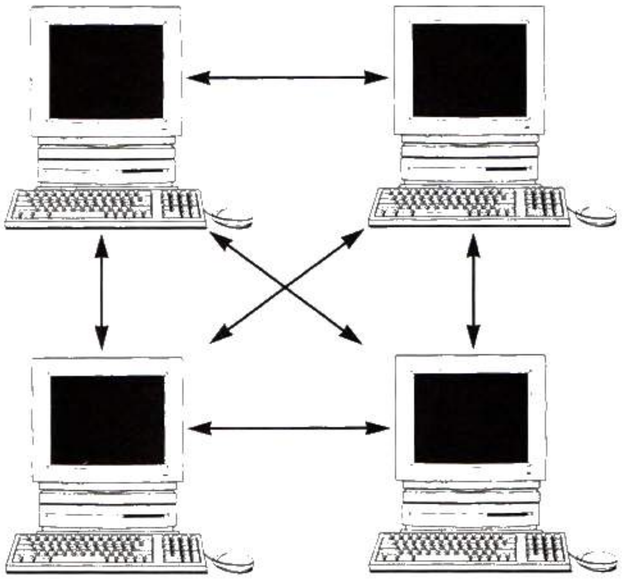


Figure 2.13 *A full connectivity scheme for a four-player game.*

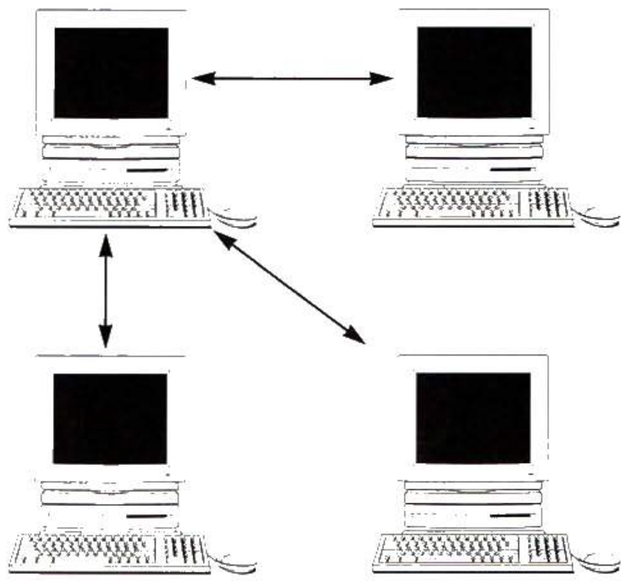


Figure 2.14 *A Client/Server scheme.*

way, all of the clients get the same information from the server. If the player on the server machine moves a playing piece, all of the players will get the information via a separate channel.

The other side of the case is when the clients have something they need to send to other players. Let's look at what happens when player 1 out of four players moves a playing piece. In this case, player 1 will send the movement information to the server. The server will immediately route the information to player 2 and player 3. The server will process the movement information from player 1 after it has been sent to players 2 and 3. This type of setup is very simple to program and administer, as we will see in Chapter 8.

Modem

The modem is even less complex. There can only be two players in a modem game. We still use the client/server protocol, but the server never has to route information to other players.

Conclusion

King's Reign is a multiplayer Windows game. It allows from two to four players to interact over the modem or Internet. The application is designed using object-oriented client/server technology.

Chapter 3

Programming Windows Applications

Although the main focus of this book is programming multiplayer games in Microsoft Windows using C++, having some background in general Windows programming is definitely a plus. This chapter should give you the background you need. You will learn about creating a basic Windows application and the skeleton necessary, as well as learn about doing Windows graphics. It will be assumed that you are familiar with the Windows environment, however. All of the code presented will be in C and can be compiled with any Windows-oriented compiler.

Microsoft Windows Fundamentals

From the very beginning, Microsoft Windows was designed to be a visual system or a GUI (graphical user interface). When you look at the screen, you will see graphical windows with additional graphics within them. The idea was to create a system something like your desk. You probably have many different types of paper on your desk right now. Some of the papers are beneath others. Not all of the papers are very important or necessary. You might have a clock on your desk as well.

In Windows, all of your work is shown on the desktop in the form of a window. There is a small window with the representation of a clock in it. When you are working with a spreadsheet, a window will contain the cells and their associated values. This window may be partially covered by another window that is being used by a word processor. If you are finished with the word processor for the moment, you can select the spreadsheet window and it will pop into view, partially covering the word processor window.

In developing Microsoft Windows applications, we have to be concerned first and foremost with the window. The operations that occur in the window are somewhat related to traditional programming, although the way in which control flows through the program will be different.

Mouse

Since we are assuming that you have used Windows before, you probably noticed that just about everything you did involved the mouse. Windows was designed to use the mouse and only occasionally the keyboard. In writing this book, I use the keyboard, but to work with my paint program, I use the mouse. I use the mouse to select bold or a different size font. This means that our game should use the mouse as efficiently as possible.

Bitmaps and Icons

Another key point to the Windows system is the use of graphics. When you have a window on the screen and minimize it, it turns into a small graphical icon. By looking at the icon, you will be able to recognize what the application is. If you are using a later version of Word for Windows, there is a two-line toolbar at the top of the window. On the toolbar are small buttons with bitmaps in them. The bitmaps are designed to clearly convey their meaning to you at a glance. One of the buttons has a bitmap of a printer on it. Without looking at the user's guide (These things come with a user's guide, it wasn't in my box...), I know that if I click on this button, my document will print.

Accessories

Just about all windows in the Windows system have accessories associated with them. By accessories, I mean menus, dialog boxes, scroll bars, and so on. These accessories ease the use of interacting with the windows. Menus give us most all of the options we can choose with this application. Dialog boxes are very important because they give us information about the current application. If you save your document, a dialog box will appear asking you to select a filename.

Flow of Control

In traditional programming, you write your program with a Main() function and other support functions. When the program is executed in say a DOS environment, the computer starts executing the program at the very first statement in the Main() function. Execution flows through the application on a line-by-line basis. When a function is called from the Main() function, control is sent to the function and does not return until it reaches a return statement or the end of the function.

In a Windows application, there is also a Main() function, as we will see shortly, but control does not flow line by line. Control is performed on an event basis. There are a number of different messages that your Windows application will be allowed to receive. When the application receives a message, it is expected to act upon the message. If you have ever worked with interrupts in DOS, you have a foundation upon which to understand event-driven systems. An interrupt acts like an event and vice versa. Every time an interrupt occurs in your computer, the thing causing the interrupt expects the computer to respond to the interrupt and do some operations. The same is true of Windows applications.

Summary

That was a short introduction to Windows and applications written for it. Now we should look at what makes up a Windows application by designing a simple Hello World program. Those with a formal background in Computer Science will know about Hello World programs. I think just about all programming languages that I learned in school and self-taught myself (the majority) started with a Hello World program.

A Simple Microsoft Windows Application

We want to design a Windows program that will create a window and display the words "Hello World" in it. Doesn't sound too tough, right? In Pascal, this is a four-line program. In traditional C, we might have five lines. Our Windows program will have considerably more lines.

As we stated earlier, one of the most important things a Windows application must do is create a window. In our first pass over a simple Windows application, this is all we will do. The program we are designing must have two components: a WinMain() function and a control function.

The WinMain() function can be loosely compared to a traditional Main() function. In our case, this function will define a window class, register the class with Windows, create and display our window, and start a message loop.

The control function must respond and act upon all messages sent to the application. The first program we create will have the control function responding to the QUIT message from Windows. The program is shown in Listing 3.1.

Listing 3.1: A Minimal Windows Application

```
#include "windows.h"


long CALLBACK _export MainWndProc(HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
```

```
{
   MSG msg;
   WNDCLASS wc;
   HWND hWnd;

   wc.style = NULL;
   wc.lpfnWndProc = MainWndProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName = NULL;
   wc.lpszClassName = "GenericWClass";


   if(!RegisterClass(&wc)) return 0;

   hWnd = CreateWindow(
     "GenericWClass",
     "Generic Sample Application",
     WS_OVERLAPPEDWINDOW,
     CW_USEDEFAULT,
     CW_USEDEFAULT,
     CW_USEDEFAULT,
     CW_USEDEFAULT,
     NULL,
     NULL,
     hInstance,
     NULL
   );

   if (!hWnd)
     return (FALSE);

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   while (GetMessage(&msg, NULL, NULL, NULL))
   {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
   }
   return (msg.wParam);
}


long CALLBACK __export MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
```

```
UINT message;
WPARAM wParam;
LPARAM lParam;
{
   switch (message)
   {
      case WM_DESTROY:
         PostQuitMessage(0);
         break;

      default:
         return (DefWindowProc(hWnd, message, wParam, lParam));
   }
   return (NULL);
}
```

The program shown in Listing 3.1 is located on the enclosed CD-ROM in the directory /netwar2/chapt3/simple.c. If you compile and execute the program, you will get the window shown in Figure 3.1.

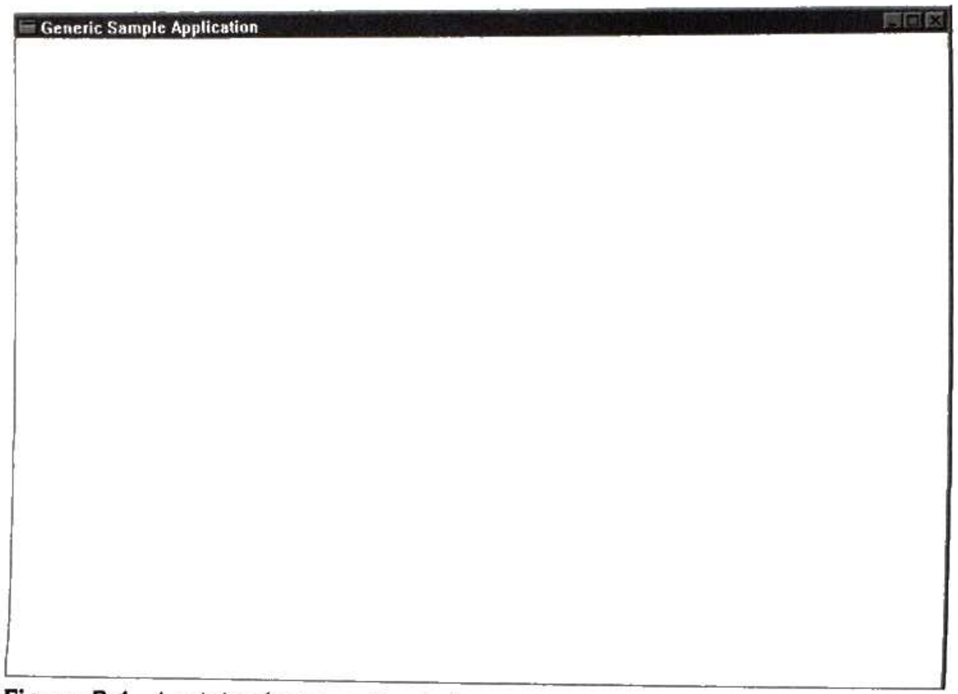


Figure 3.1 *A minimal program's window.*

Before going any further, we should go through the program line by line so that you understand how the program works. The very first line of the program listing is the statement # include <windows.h>. This include file contains all of the prototypes, macros, and definitions for Windows programs. In this file, you will find the required format for the WinMain() function as well as the definition of the data type HWND. It must be included in *all* Windows programs.

As we stated above, our minimal Windows program must have a control function. The statement *long CALLBACK _export MainWndProc(HWND, UINT, WPARAM, LPARAM);* declares the control function. The actual code for the function will occur later in the function.

Next we encounter the start of the WinMain() function. You will notice that four parameters are sent to this function when the program is executed. The parameters are *hInstance, hPrevInstance, lpCmdLine,* and *nCmdShow.* The *hInstance* parameter is a handle with a value that indicates which instance of this program the currently running application is. If you have used Windows before, you know that many programs allow you to execute them more than once. Each execution of a Windows program is called an instance. Many times an application will need to know which instance it is so that information meant for one application is not received by another. The second parameter, *hPrevInstance,* is the value of the previous instance of this application. The *lpCmdLine* parameter would contain a list of all command-line parameters specified when the application executed. Command-line parameters are rarely used in Windows applications, but they can be, so provision must be made for them. The last parameter, *nCmdShow,* is a value used to determine how the window for this application should be displayed when the program is first executed. You might want your window to be minimized or maximized upon startup.

WinMain()

Now we can go into the WinMain() function itself. As stated above, the first thing the WinMain() function must do is define a window class. Although this might sound intimidating, Windows does a good deal of the work for us. First off, we need to have a variable declared of the type *WNDCLASS.* In our program, this variable is called *wc*. The fields available in the *WNDCLASS* type are:

wc.style	the type of window created
wc.lpfnWndProc	the callback control function
wc.cbClsExtra	extra class information
wc.cbWndExtra	extra window information
wc.hInstance	the instance of this application
wc.hIcon	the icon to use for this application
wc.hCursor	the cursor to use for this application
wc.hbrBackground	the color of the window background
wc.lpszMenuName	name of window menu
wc.lpszClassName	name of the window class

I will refer you to the Windows programming reference guides for more information on each of the fields of the *WNDCLASS* type, since we aren't going to be doing much with this type when we start our game development in C++. Just be aware that this type does exist and is used to register the window class with Windows.

After the fields of the class variable have been filled, it is time to actually register the window. The code to do this is *if(!RegisterClass(&wc)) return 0*. If things go as planned, the class has been registered and it's time to create a window of the class we registered. The code used to create the window is:

```
hWnd = CreateWindow(
   "GenericWClass",          /* name of the window class */
   "Generic Sample Application",        /* title of the window */
   WS_OVERLAPPEDWINDOW,         /* type of window */
   CW_USEDEFAULT,               /* upper-left corner x */
   CW_USEDEFAULT,               /* upper-left corner y */
   CW_USEDEFAULT,               /* lower-right corner x */
   CW_USEDEFAULT,               /* lower-right corner y */
   NULL,                                /* handle of the parent window */
   NULL,                                /* handle of main menu */
   hInstance,                           /* handle of creator */
   NULL                                 /* additional information pointer*/
);
```

Again, I will refer you to the programming reference manuals for Windows for more information on the CreateWindow() function and its parameters.

A call to CreateWindow() will return a handle to the application's window. The handle is kept in the variable *hWnd*. A quick check of the variable will tell us if a window was successfully created or not.

```
if (!hWnd)
 return (FALSE);
```

If *hWnd* is not something other than a 0, then the window was not created and we should not execute the program.

With the window successfully created, it can be displayed on the screen using the function ShowWindow(hWnd, nCmdShow);. This function displays the window using the window style parameter originally sent to the WinMain() function. After the window is displayed, the function UpdateWindow(hWnd)(); is called. This function sends an update message to the window. In just about all Windows programs, this is a necessary function call, but not in our minimal program.

The last part of the WinMain() function is very important. It is the message loop and is required in all Windows programs. Remember that all Windows applications execute through an event-driven sequence. This message loop does the actual receiving and processing of messages to this application. Our application must use the function GetMessage() to retrieve messages from an application message queue that stores all messages coming to this function. The code for the message loop is:

```
while (GetMessage(&msg, NULL, NULL, NULL))

{

 TranslateMessage(&msg);

 DispatchMessage(&msg);
}
```

A message in the Windows environment is made up of several parts. A quick look at the MSG data type shows a structure like

```
typedef struct tagMSG
{
  HWND hwnd; /* the window the message is for */
  UINT message; /* message number */
  WPARAM wParam; /* message-dependent information */
  LPARAM lParam; /*message-independent information */
  DWORD time; /* time message posted to application queue */
  POINT pt; /*x, y location of mouse */
} MSG;
```

When we talk about a message in Windows, we are not talking about a little note with words on it. A message is just a value that represents a particular function. For instance, the message WM_DESTROY tells the current application to destroy its window and terminate. This message is just a value that is stored in the *message* field. All messages used by your application and Windows itself should have different values.

If we look back at the message loop code above, we see a call to the function GetMessage(). If this function returns a value of 0, then the user has closed the window and exited the program. In the details of the GetMessage() function, the system will check the application's message queue for any messages. If no messages are found for this application, control is returned to the Windows system. If a message is found for this application, a call is made to the function TranslateMessage(). This function takes virtual key codes on the keyboard and translates them in character messages. The message is then passed to the DispatchMessage() function. This function will pass the message to the program's control function we defined earlier.

The Control Function

This brings us directly into the control function. The control function is defined as

```
long CALLBACK _export MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
{
   switch (message)
   {
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
```

```
        default:
            return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}
```

As you can see, the main body of the function is just a *switch* statement controlled by the value in the message field of the message currently being processed. When the function is called, the parameters are filled with the window this message is for, the message itself, and the two additional information fields. The function uses this information to determine what the message is and what to do with it.

In our case, we are only looking for one message. All other messages are passed back to the Windows system for processing via the *default* statement. Thus, if the value of *message* is equal to WM_DESTROY, we post a quit message to the current application and stop processing. That's all there is to Windows message processing. Well, it can get a bit more complex, but as we will see in Chapter 4, we don't have to worry about all of this detail.

Adding Hello World

If you ran the program above, you will notice that nothing is printed in the window. Let's add some text. When advancing our program to actually doing something in a window, we have to take a look at the mechanism that Windows provides to do this. The mechanism is called a device context (DC). Figure 3.2 shows where the device context fits in the Windows system.

As you can see from Figure 3.2, the device context is a link between your application and a device driver. Device drivers are usually provided by the manufacturer of a video card. The device driver has specific routines for driving the video card.

For our purpose, writing to a window, the Windows system maintains a cache of device contexts. We will have to request a device context and release it after we are finished with it. In order to request and release a device context, we have to interact with Windows' graphical device interface (GDI). The GDI is a set of functions that act upon the device context. There are entirely too many functions in the GDI to list here, so check a Windows programming reference

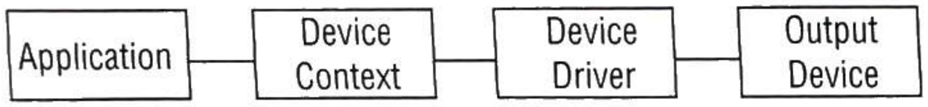


Figure 3.2 *The device context.*

guide for a list. Using a device context means your application does not have to concern itself with the output device. The device context will contact the device driver on your behalf and handle any specific features.

So, now let's put some text on the screen. We will put the text on the screen when the user presses the X key on the keyboard. Add the following text to the *switch* statement in the MainWndProc() function:

```
case WM_CHAR:
             hdc = BeginPaint(hwnd, &PaintStruct);
             GetClientRect(hwnd, &rect);
             DrawText(hdc, "Hello World", -1, &rect, DT_SINGLELINE );
             EndPaint(hwnd, &PaintStruct);
             break;
```

At the start of the same function, add the following declarations:

```
HDC             hdc;
RECT            rect;
PAINTSTRUCT     PaintStruct;
```

Now you can recompile and execute the program. Press any key on the keyboard to see the "Hello World" statement appear in our window. You can find the new source code in the directory /netwar2/chapt3/hello.c.

Let's take a look at the statements we added to the program. The first thing we did was add a *case* statement to the message *switch* routine. When a message matching WM_CHAR is sent to our application, we will execute the code after the *case* statement. The WM_CHAR is sent to our application when the Windows system software determines that we pressed a key on the keyboard.

Once we have determined that a key was pressed, we execute the line *hdc = BeginPaint(hwnd, &PaintStruct);*. The function BeginPaint() is a Windows GDI function. Its purpose is to prepare a window for painting. This includes filling the second parameter with information about the painting and returning a device context handle to the caller. We take the device context handle and put it into the local variable *hdc*.

By calling the BeginPaint() function, we have asked for permission to draw on our application's window. The function retrieves a device context for us and validates the client region for drawing. What a minute. What do we mean by "validates the client region"? If you look at a Windows application window, you will see that the majority of the window is a, usually, white background area. This is called the client region. By validating this region, Windows gives our application the ability to draw in the specific region. There are areas on the window that you are not usually allowed to draw on. These areas include the menu, scroll bars, and so on.

Now that the window is ready for painting, we make a call to the function GetClientRect(). This function obtains the drawable area of our window and puts the coordinates in the RECT parameter supplied to the function. The coordinates will be used in the next statement.

The next statement does the actual drawing of our text in the window. The statement is *DrawText(hdc, "Hello World", -1, &rect, DT_SINGLELINE);*. This function is part of the Windows GDI and acts directly on a device context. In fact, the very first parameter to the function is the handle of the device context it should use.

The second parameter to the DrawText() function call is the text to be drawn on the screen. The third parameter specifies whether or not the text drawn will be terminated with a NULL character. The fourth parameter is the rectangle that specifies the allowable drawing area in the window. The last parameter is a flag. This flag dictates how the text will be drawn on the window. Some of the options available are DT_CENTER, which would center the text horizontally, and DT_VCENTER, which would center the text vertically. The different options available are ORed together using the C "|" symbol.

Once this statement is executed, the text will be drawn on the screen. The last thing we need to do is let the system know that we are finished drawing on the screen. This is done with the EndPaint() function call.

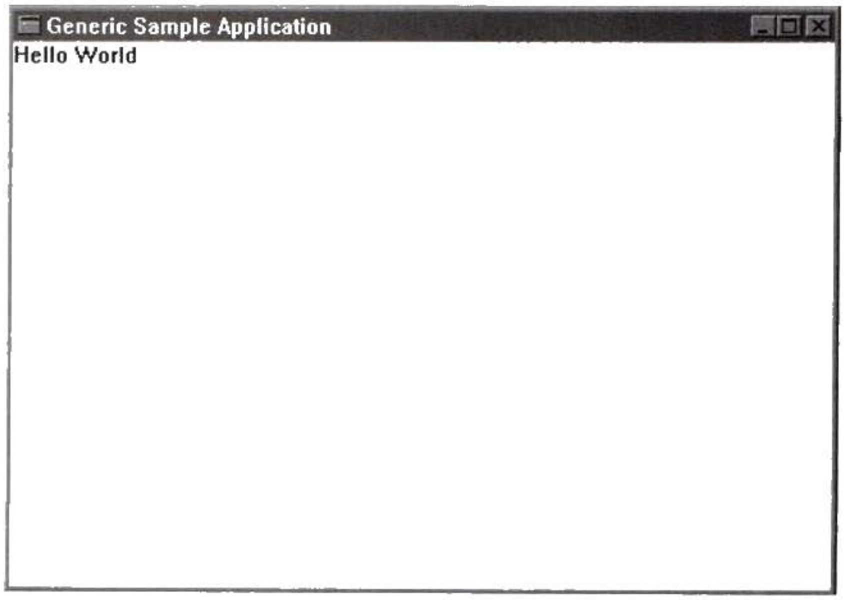
Disappearing Text

If you executed the program above, you may have noticed that nothing has appeared on the screen. No matter which key you press, "Hello World" never appears. The explanation for this is found in the way Windows handles redrawing the screen.

Each time you press a key, Windows validates the region to draw in. Once you are finished drawing, Windows invalidates the region. This causes a WM_PAINT message to be sent to your application. Since you don't do anything with this message, it is sent to Windows to handle. Windows handles it by redrawing everything in the application's window. In this case, Windows just redraws the background, which covers up our Hello World text. We don't see this happening because Windows redraws the screen so fast.

You can verify that we are indeed capturing the WM_CHAR message and drawing on the screen by putting a *MessageBeep(1);* statement after the *WM_CHAR* statement. Recompile and execute the program. You will hear a beep each time you press a key.

So what do we do? Since Windows is going to redraw our screen using a WM_PAINT message, let's put our "Hello World" message in a WM_PAINT case. If you added the *MessageBeep(1);* statement, delete it. Now change the



Hello World

Figure 3.3 *Output of the "Hello World" program.*

WM_CHAR message to WM_PAINT and recompile the program. Execute the program to see "Hello World" at the top of the screen. The output should look like Figure 3.3.

Adding Graphical Operations to Your Windows Application

The previous program was fairly simple. It's time to look at something more complex. This time we are going to create a couple of programs that will illustrate how to do Windows graphics.

We will be using the same minimal program as above, but with several added components. The first thing we need to do is respond to the mouse. The program we are going to start with is shown in Listing 3.2.

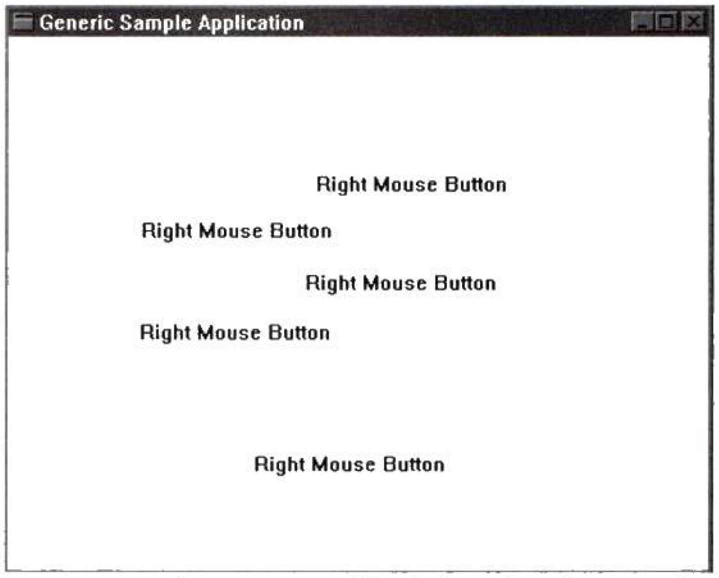
Listing 3.2: A Minimal Program That Responds to Mouse Commands

```
#include "windows.h"
#include "string.h"

char str[80];

long CALLBACK __export MainWndProc(HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
   MSG msg;
   WNDCLASS wc;
   HWND hWnd;

   wc.style = NULL;
   wc.lpfnWndProc = MainWndProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName = NULL;
```

```
    wc.lpszClassName = "GenericWClass";

    if(!RegisterClass(&wc)) return 0;

    hWnd = CreateWindow(
      "GenericWClass",
      "Generic Sample Application",
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL
     );

     if (!hWnd)
       return (FALSE);

     ShowWindow(hWnd, nCmdShow);
     UpdateWindow(hWnd);

     while (GetMessage(&msg, NULL, NULL, NULL))
     {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
       }
       return (msg.wParam);
}


long CALLBACK __export MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
{
    HDC hdc;
    PAINTSTRUCT PaintStruct;

      switch (message)
      {
        case WM_DESTROY:
          PostQuitMessage(0);
          break;

        case WM_RBUTTONDOWN:
          hdc = GetDC(hWnd);
          strcpy(str, "Right Mouse Button");
      TextOut(hdc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));
      ReleaseDC(hWnd, hdc);
      break;

    default:
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (NULL);
}
```

You should compile and execute the program listed in Listing 3.2. You can find the source code on the CD-ROM in the directory /netwar2/chapt3/ mouse.c and mouse.exe. Once you have the program running, put the cursor in the window's client region and click the right mouse button. The words "Right Mouse Button" should appear at the location the mouse cursor is location. Now move the cursor around the window, clicking the right mouse button. Many different "Right Mouse Button" strings will appear. The program's output might look like Figure 3.4.

You will recall that we mentioned above that each time we draw something on the screen a WM_PAINT message is sent to the application. In order for



```
Right Mouse Button
Right Mouse Button
Right Mouse Button
Right Mouse Button
Right Mouse Button
```

Figure 3.4 *The output caused by clicking the right mouse button.*

the things that we have drawn on the screen to remain on the screen, we have to redraw them within the WM_PAINT message case. Well, that isn't entirely true. The BeginPaint() and EndPaint() functions issue a WM_PAINT message to the application, but we don't use these functions in our new program.

To see why, let's look at what happens when we click the right mouse button. When the right mouse button is clicked, the Windows system sends a WM_RBUTTONDOWN message to our application. To respond to this message, we have a *case* statement in our control function that will recognize the WM_RBUTTONDOWN message.

When the message appears, the statement *hdc = GetDC(hWnd);* is executed. This statement asks the Windows system for a device context for the window associated with the *hWnd* handle. The *hWnd* handle will always be associated with our window because the Windows system passes it to us when responding to messages. You will see this variable in the parameter list of the control function. If a device context is available, the local variable *hdc* will be set to the device context's handle. We are now ready to write to the screen.

The statement *strcpy(str, "Right Mouse Button");* copies the text we want to output to the global variable *str*. We output the text with the statement *TextOut(hdc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));*. This statement needs to be looked at in detail. First, we see that the function is passed the device context we are using. The second and third parameters represent the X and Y position on the screen our text should be written to. The fourth parameter is our text, and the fifth is the length of the text to output.

After the text is written to the window, we have to release the device context that we allocated earlier with the statement *ReleaseDC(hWnd, hdc);*. This explanation of the code doesn't explain why the system doesn't repaint the window and erase all of the text on it. The reason that this doesn't happen is because the GetDC() and ReleaseDC() function calls do not tell Windows to send a WM_PAINT message to the application. If we wanted to, we could send the repaint message ourselves and we will in the next section. Right now, the only way to get the system to send a WM_PAINT message to our application is through resizing, moving, or minimizing the window. Try it and see what happens.

Another thing you should know about the GetDC() function is that unlike the BeginPaint() function, it does not take control of the entire client region of the window. It will only use as much of the client region as it needs to fulfill your paint commands. If you right click the mouse button on the screen and then move the mouse just a little up the screen and click the right mouse button again, you will see that the text overwrites the previous text with just a slight white border around it. The device context automatically calculates the right amount of screen area to use.

In most cases, we will want to use the GetDC() function instead of the BeginPaint() function so that we use as little of the client region as possible.

Generating a WM_PAINT Message

Why would be want to generate a WM_PAINT message? Well, one reason would be to blank the screen. As an example, take a look at the control function in Listing 3.3.

Listing 3.3: Using InvalidateRect() to Clear the Screen

```
#include "windows.h"
#include "string.h"

char str[80];

long CALLBACK _export MainWndProc(HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
   MSG msg;
   WNDCLASS wc;
   HWND hWnd;

   wc.style = NULL;
   wc.lpfnWndProc = MainWndProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

```
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = GetStockObject(WHITE_BRUSH);
     wc.lpszMenuName = NULL;
     wc.lpszClassName = "GenericWClass";

     if(!RegisterClass(&wc)) return 0;

     hWnd = CreateWindow(
        "GenericWClass",
        "Generic Sample Application",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
     );

     if (!hWnd)
        return (FALSE);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    while (GetMessage(&msg, NULL, NULL, NULL))
    {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
     }
     return (msg.wParam);
}

long CALLBACK __export MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
{
   HDC hdc;
   PAINTSTRUCT PaintStruct;
   RECT rect;

   switch (message)
   {
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
      case WM_RBUTTONDOWN:
         hdc = GetDC(hWnd);
         strcpy(str, "Right Mouse Button");
         TextOut(hdc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));
         ReleaseDC(hWnd, hdc);
         break;

      case WM_LBUTTONDOWN:
         InvalidateRect(hWnd, NULL, TRUE);
         break;

      case WM_PAINT:
                 hdc = BeginPaint(hWnd, &PaintStruct);
           GetClientRect(hWnd, &rect);
           DrawText(hdc, "Hello World", -1, &rect, DT_SINGLELINE );
           EndPaint(hWnd, &PaintStruct);
         break;

      default:
         return (DefWindowProc(hWnd, message, wParam, lParam));
   }
   return (NULL);
}
```

This program uses the function InvalidateRect() to clear the screen when the left mouse button is clicked. The listing can be found in the directory /netwar2/chapt3/mouseclr.c and mouseclr.exe. When you execute the program, right click the mouse a few times and then click the left mouse button. The screen will be cleared.

Look at the InvalidateRect() function in the control function in Listing 3.3. The first parameter in this statement is the window that should have its client region invalidated. When we say invalidate, we mean that it needs to be repainted. This causes a WM_PAINT message to be sent to our application. The second parameter is a rectangle of the region that should be cleared. If a NULL rectangle is used, the entire screen is cleared. The last parameter indicates whether or not the background should be cleared.

Now what do you think would happen if we put the following statements into a WM_PAINT case statement in our control function?

```
  hdc = BeginPaint(hwnd, &PaintStruct);
GetClientRect(hwnd, &rect);
```

```
    DrawText(hdc, "Hello World", -1, &rect, DT_SINGLELINE );
    EndPaint(hwnd, &PaintStruct);
```

Well, if you execute the program /netwar2/chapt3/mousclr2.exe, you will see the result. Click the right mouse button a few times and then click the left button. Hey, "Hello World" is back. This is because we added our own WM_PAINT message. Therefore, if there is *anything* that you want to always be in the window, you must put it into the WM_PAINT message *case* statement. This causes the information to be redrawn whenever the window is repainted.

Doing a Graphic

Now that we are moving along, let's try to draw a graphical rectangle in the client region of our window. The new program is in Listing 3.4.

Listing 3.4: A Program to Draw a Rectangle

```
#include "windows.h"
#include "string.h"

char str[80];

long CALLBACK _export MainWndProc(HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
   MSG msg;
   WNDCLASS wc;
   HWND hWnd;

   wc.style = NULL;
   wc.lpfnWndProc = MainWndProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName = NULL;
   wc.lpszClassName = "GenericWClass";

   if(!RegisterClass(&wc)) return 0;
```

```
    hWnd = CreateWindow(
      "GenericWClass",
      "Generic Sample Application",
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL
    );

    if (!hWnd)
      return (FALSE);
      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);

      while (GetMessage(&msg, NULL, NULL, NULL))
      {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
  return (msg.wParam);
}

long CALLBACK __export MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
{
    HDC hdc;
    PAINTSTRUCT PaintStruct;
    RECT rect;

      switch (message)
      {

         case WM_DESTROY:
            PostQuitMessage(0);
            break;

         case WM_RBUTTONDOWN:
            hdc = GetDC(hWnd);
            strcpy(str, "Right Mouse Button");
            TextOut(hdc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));
            ReleaseDC(hWnd, hdc);
            break;
        case WM_LBUTTONDOWN:
           InvalidateRect(hWnd, NULL, TRUE);
           break;

        case WM_PAINT:
           hdc = GetDC(hWnd);
           Rectangle(hdc, 10,10,100,100);
           ReleaseDC(hWnd, hdc);
           break;

        default:
           return (DefWindowProc(hWnd, message, wParam, lParam));
     }
     return (NULL);
}
```

You can find the program in /netwar2/chapt3/rect.c and rect.exe. When you execute the program, a rectangle will be drawn on the screen as shown in Figure 3.5. Just as in the case of displaying text, we allocate a device context and use a GDI function called Rectangle() to draw our rectangle.

Handling WM_PAINT Messages

You may have noticed that we have a problem with the programs we have been developing. In order to keep a window up-to-date visually, we have to repaint the window each time a WM_PAINT message is received. Now with

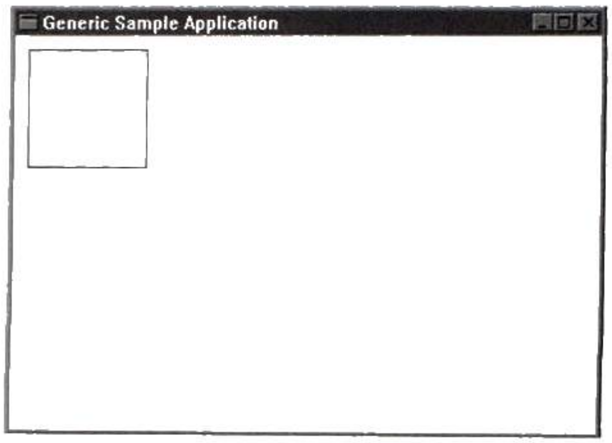


Figure 3.5 *The rectangle on the screen.*

our simple programs, we don't receive the WM_PAINT message much, but each time our application's window is covered up in any way, and then is put back into view, a WM_PAINT message will occur.

The question we have to answer is how do we keep track of all of the information that has been put on the screen? Do we keep a record of all graphics operations in a log and replay the log each time the window gets the WM_PAINT message? There are several ways to handle this problem, but the one technique directly supported by Windows is to use a memory device context.

A memory device context is a device context that exists in memory and is fully compatible with a normal device context. When we have any graphical operations to perform, we simply use the memory device context instead of a standard device context. When a WM_PAINT message is received from Windows, a quick copy is made from the memory device context to the window client region. In addition to the memory device context, we use a bitmap to hold the actual graphics that are intended for the screen. The bitmap is created to be compatible with the memory device context as well as a standard device context.

So, let's see how this memory device context works. Listing 3.5 is a program that uses a memory device context to draw to a window.

Listing 3.5: A Program for Memory Device Context Drawing

```
#include "windows.h"
#include "string.h"

char str[80];
HDC memorydc;
int screenx, screeny;
HBITMAP hscreen_bitmap;
long CALLBACK __export MainWndProc(HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{

   MSG msg;
   WNDCLASS wc;
   HWND hWnd;
   wc.style = NULL;
   wc.lpfnWndProc = MainWndProc;
   wc.cbClsExtra = 0;
   wc.cbWndExtra = 0;
   wc.hInstance = hInstance;
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
   wc.hbrBackground = GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName = NULL;
   wc.lpszClassName = "GenericWClass";


   if(!RegisterClass(&wc)) return 0;

   hWnd = CreateWindow(
      "GenericWClass",
      "Generic Sample Application",
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL
   );

   if (!hWnd)
      return (FALSE);

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   while (GetMessage(&msg, NULL, NULL, NULL))
   {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
   }
   return (msg.wParam);
}


long CALLBACK __export MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
UINT message;
WPARAM wParam;
LPARAM lParam;
{
   HDC hdc;
```

```
    PAINTSTRUCT PaintStruct;
    RECT rect;
    HBRUSH hbrush;
    switch (message)
    {
       case WM_CREATE:
          screenx = GetSystemMetrics(SM_CXSCREEN);
          screeny = GetSystemMetrics(SM_CYSCREEN);

          hdc = GetDC(hWnd);
          memorydc = CreateCompatibleDC(hdc);
          hscreen_bitmap = CreateCompatibleBitmap(hdc, screenx, screeny);

          SelectObject(memorydc, hscreen_bitmap);
          hbrush = GetStockObject(WHITE_BRUSH);
          SelectObject(memorydc, hbrush);

          PatBlt(memorydc, 0, 0, screenx, screeny, PATCOPY);

          ReleaseDC(hWnd, hdc);
          break;

       case WM_DESTROY:
          PostQuitMessage(0);
          break;

       case WM_RBUTTONDOWN:
          strcpy(str, "Right Mouse Button");
          TextOut(memorydc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));
          break;

       case WM_LBUTTONDOWN:
          InvalidateRect(hWnd, NULL, TRUE);
          break;

       case WM_PAINT:
          hdc = BeginPaint(hWnd, &PaintStruct);
          BitBlt(hdc, 0, 0, screenx, screeny, memorydc, 0, 0, SRCCOPY);
          EndPaint(hWnd, &PaintStruct);
          break;

     default:
          return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}
```

This program can be found in the directory /netwar2/chapt3/virtual.c and virtual.exe. When you execute the program, move the mouse around the window and click the right mouse button a couple of times. You will find that nothing appears on the screen. Now click the left mouse button once. All of the right mouse button messages instantly appear on the screen. This is certainly different from the other programs. Let's see how the memory device context was used in this program.

If you look in the control function, you will find a *case* statement for the message WM_CREATE. Every Windows application will be sent a WM_CREATE message upon startup. It is only sent once, so it is a good place to put statements that you only want executed once at startup.

The code for this message starts with the statement:

```
screenx = GetSystemMetrics(SM_CXSCREEN);
screeny = GetSystemMetrics(SM_CYSCREEN);
```

The variables *screenx* and *screeny* are declared as standard INT variables. Each of the variables is assigned the X and Y coordinate value for our window. In other words, once the window for our application is created, the GetSystemMetrics() function will return, depending on its parameter, the X or Y maximum for the window. We will be using these variables shortly.

The next statement should look familiar

```
hdc = GetDC(hWnd);
```

We need to get a standard device context before we create a memory device context. After obtaining our device context, the statement

```
memorydc = CreateCompatibleDC(hdc);
```

is issued. This statement creates a memory device context that is compatible with the device context sent as a parameter to the CreateCompatibleDC() function. The handle for our memory device context is stored in the variable *memorydc*, which is declared as a global HDC. Next we have to obtain a bitmap with the statement

```
hscreen_bitmap = CreateCompatibleBitmap(hdc, screenx, screeny);
```

The bitmap is created with the CreateCompatibleBitmap() function. This function creates a bitmap that has a one-to-one relationship with our window and makes sure that it is compatible with the device context sent as parameter one. Notice that parameters two and three are the variables we put the size of our application's window in. These parameters will give us a bitmap that is exactly the same size as our window. The handle to the bitmap is stored in the global variable *hscreen_bitmap*.

What we are doing is creating a bitmap that is associated with a memory device context. We are going to do all screen draws to the memory bitmap and copy it to the window when necessary. This allows us to concentrate all window writing to the WM_PAINT message handler as we will see shortly.

After our bitmap is created, we associate the bitmap with the memory device context.

```
SelectObject(memorydc, hscreen_bitmap);
```

This causes all writes to the memory device context to be put on the bitmap. Before we can quit, we have to make sure that the background of the memory bitmap matches the background of our window. You will recall from the WinMain() function that we assigned a WHITE_BRUSH object to the background of our window. We can do the same thing with the statements

```
hbrush = GetStockObject(WHITE_BRUSH);
SelectObject(memorydc, hbrush);
```

What we have done is create a standard WHITE_BRUSH brush. This is a GDI function. The brush is then associated with the memory device context. The bitmap can now be painted white with the statement

```
PatBlt(memorydc, 0, 0, screenx, screeny, PATCOPY);
```

Finally, we can release the standard device context we allocated at the start of the WM_CREATE message handler

```
ReleaseDC(hWnd, hdc);
```

With the memory device context created, we can begin using it. Here is code that handles a right mouse button click:

```
case WM_RBUTTONDOWN:
    strcpy(str, "Right Mouse Button");
```

```
        TextOut(memorydc, LOWORD(lParam), HIWORD(lParam), str, strlen(str));
        break;
```

As you can see by looking back at Listing 3.4, the code is a little smaller. No longer do we need to allocate a device context. The TextOut() function called writes directly to the *memorydc* instead of the *hdc*. A TextOut() function does not cause the window to be redrawn, so you won't see anything on the screen. All of the text has been drawn on a memory bitmap and not the window.

In order to get the bits in the memory bitmap on the screen, we turn to the left mouse button handler code. It is

```
    case WM_LBUTTONDOWN:
        InvalidateRect(hWnd, NULL, TRUE);
        break;
```

The only thing in this code is a call to the function InvalidateRect(). As we already know, this causes a WM_PAINT message to be sent to the application. As you might have guessed, the copy of the memory bitmap to the window will be done in the WM_PAINT code. That code is:

```
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &PaintStruct);
        BitBlt(hdc, 0, 0, screenx, screeny, memorydc, 0, 0, SRCCOPY);
        EndPaint(hWnd, &PaintStruct);
        break;
```

This code starts with the BeginPaint() function. We are going to copy the memory bitmap to the window and we know they are the same size, so we had better get permission to draw on the entire client region of the window. The second statement does the actual copying of the memory bitmap to the window. The first parameter is the device context to draw *to*. The second through fourth parameters are the starting and ending points of the bitmap to draw. The fifth parameter is the device context to draw *from*. The sixth and seventh parameters are the upper-left corner of the source bitmap. The last parameter is a flag that indicates how the bitmap should be copied.

As you can see from this program, all of the actual drawing to the window is done through the WM_PAINT message code. No matter what happens to the window, even if it is covered up or minimized, it will be updated properly without having to keep a log or replay previous paint commands.

Summary

Programming in Windows using C requires the programmer to keep track of and account for many of the operations of the application. Using a standard skeleton program, the initialization of the Windows program can be simplified. Since using a skeleton program is a popular option for Windows programming, Microsoft itself developed a set of functions and objects that are intended to cut down on the standard Windows code and allow programmers to concentrate on the specifics of their own code. This set of functions and objects is called the Microsoft Foundation Classes.

Chapter 4

Programming With Visual C ++

We've seen how to program for Windows in the previous chapter. This was and still is the standard for some ways of developing Windows applications. As the programming world advanced, C was no longer considered the language of choice. The new language that developers are using is C++. Once mastered, C++ is a more natural way of programming.

Seeing the advanced use of C++, Microsoft developed and continues to refine the Visual C++ product. In this chapter, we will discuss the components available in Visual C++, look at the Microsoft Foundation Classes, which are an integral part of Visual C++, and build some programs using what we have learned.

Visual C++

Visual C++ is a software development package for creating applications designed to execute on the Microsoft Windows family of operating environments and systems. The complete system is integrated and graphical, allowing for increased productivity. Two versions of the package are available: 16-bit and 32-bit. The 16-bit version, currently in rev 1.53, is designed to produce 16-bit Windows applications that are compatible with all Windows systems including Windows 3.x. The 32-bit version, currently in rev 4.0, is designed to produce 32-bit Windows applications, which are designed to execute on Windows NT, Windows 95, and systems running WIN32.

The package includes a C/C++ compiler, a linker, and a graphical work area. The work area is called the Visual WorkBench and allows for editing of C/C++ code. From the WorkBench, you can execute any of the Visual C++ applications.

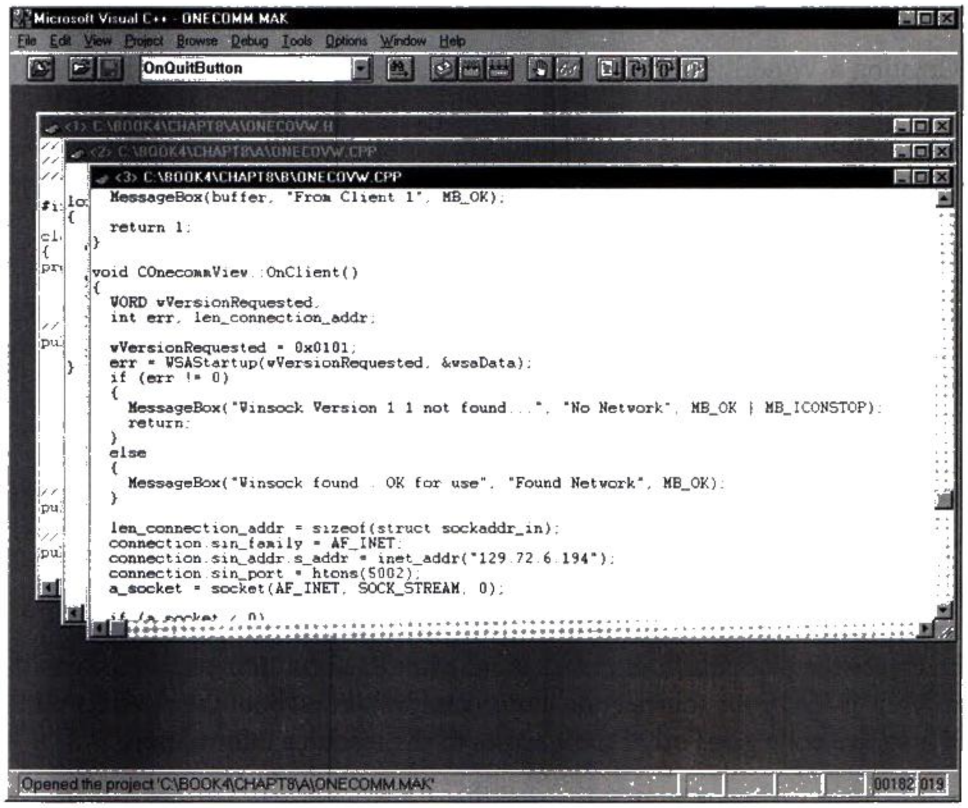
The real power behind the Visual C++ package is the Microsoft Foundation Classes library (MFC). The purpose of the MFC library is to encapsulate most all of the functions of the Software Development Kit normally used to develop Windows applications. The encapsulation allows a developer to use object-oriented programming to develop Windows applications in significantly less time than that required by C and the SDK.

Visual WorkBench

The Visual WorkBench is like a workbench. You have a surface to work on and a series of tools available to you. The Visual WorkBench is shown in Figure 4.1.

In Figure 4.1, you can see a piece of C++ code is on the working surface of the bench. At the top of the Visual WorkBench are the tools available to the developer. The tools include:

- **AppWizard:** An application generator that produces a simple skeleton for a Windows application. The skeleton is used as a base that is enhanced with your own code.
- **App Studio:** A resource generator and editor. This graphical tool allows you to create and edit dialog boxes, menus, bitmaps, icons, string tables, and accelerator-key tables. A set of tools makes the creation of these required Windows components easy.
- **ClassWizard**: A class generator and editor. Using the ClassWizard allows you to create and add classes to your application easily, as well as handle classes already in your application. The ClassWizard makes handling messages a snap.
- **Debugger:** A fully functional debugger is available to help find errors in your code. You can set breakpoints, step through your code, and perform other functions as well.


Microsoft Visual C++ - ONECOMM.MAK
File Edit View Project Browse Debug Tools Options Window Help
OnQuitButton
<1> C:\BOOK4\CHAPT8\A\ONECOVW.H
<2> C:\BOOK4\CHAPT8\A\ONECOVW.CPP
<3> C:\BOOK4\CHAPT8\B\ONECOVW.CPP

```
  MessageBox(buffer, "From Client 1", MB_OK);

  return 1;
}

void COnecommView::OnClient()
{
  WORD wVersionRequested;
  int err, len_connection_addr;

  wVersionRequested = 0x0101;
  err = WSAStartup(wVersionRequested, &wsaData);
  if (err != 0)
  {
    MessageBox("Winsock Version 1.1 not found...", "No Network", MB_OK | MB_ICONSTOP);
    return;
  }
  else
  {
    MessageBox("Winsock found...OK for use", "Found Network", MB_OK);
  }

  len_connection_addr = sizeof(struct sockaddr_in);
  connection.sin_family = AF_INET;
  connection.sin_addr.s_addr = inet_addr("129.72.6.194");
  connection.sin_port = htons(5002);
  a_socket = socket(AF_INET, SOCK_STREAM, 0);
```

```
Opened the project 'C:\BOOK4\CHAPT8\A\ONECOMM.MAK'    00182 019
```

Figure 4.1 *The Visual C++ WorkBench.*

- **Source Browser:** Once your application has been built, you can view your classes and objects in a hierarchy tree.
- **Diagnostic Tools:** The Visual C++ package includes various diagnostic tools: Spy for observing Windows messages, HeapWalker for peering into your memory, HC31 for compiling help files, Stress for limiting available memory, and a profiler to find the slow parts of your code.
- **Help:** A comprehensive help system gives you information on Visual C++, the Microsoft Foundation Classes, the Windows SDK, and the C and C++ languages.

AppWizard

Creating a Windows application was never easier with the use of the AppWizard. This application generator will create a complete SDI or MDI Windows application. The AppWizard gives you the ability to select a large number of options that customize the code generated. If you want a toolbar, the generator can create it. If you want print and preview ability, AppWizard is happy to include it. You can even include OLE2 capability.

The code created is easily edited and includes comments so that you can follow what the code generator produced. The code is compatible with all of the other Visual C++ tools.

AppStudio

The AppStudio tool is a comprehensible creator and editor for the resources that you include in your Windows application. What would a Windows application be without a couple of dialog boxes? The AppStudio allows you to create the resources and automatically inserts them into your resource file as well as into your source code. Putting information about the resources into the source code gives other tools access to the resource information.

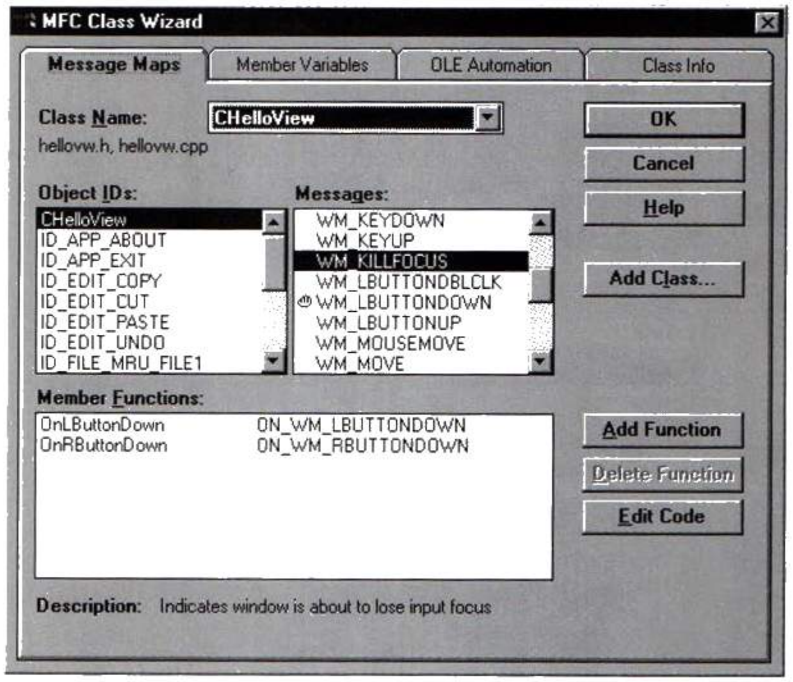
ClassWizard

The AppWizard and AppStudio tools coordinate together to create a complete Visual C++ application. The resulting code has information embedded in it that the ClassWizard application can use along with the Microsoft Foundation Class library to manage all of the classes.

Four main areas are available in the ClassWizard. These are message maps, member variables, OLE Automation, and class info.

Message Maps

When the AppWizard creates your application, only few messages are handled. To handle the rest, you can use the message map feature of the ClassWizard to identify and insert code automatically into your application for all of the messages available for a specific class. Figure 4.2 shows an example of a View class and the messages available.



Class Name: CHelloView
hellovw.h, hellovw.cpp

Object IDs:
CHelloView
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_COPY
ID_EDIT_CUT
ID_EDIT_PASTE
ID_EDIT_UNDO
ID_FILE_MRU_FILE1

Messages:
WM_KEYDOWN
WM_KEYUP
WM_KILLFOCUS
WM_LBUTTONDBLCLK
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_MOUSEMOVE
WM_MOVE

Member Functions:

OnLButtonDown	ON_WM_LBUTTONDOWN
OnRButtonDown	ON_WM_RBUTTONDOWN

Description: Indicates window is about to lose input focus

Figure 4.2 *The ClassWizard message map option.*

By identifying the message and clicking on the Add Function button, the code for handling the message is added to the view object's header and source file. All you need to do is click on the Edit Code button and add your specific code.

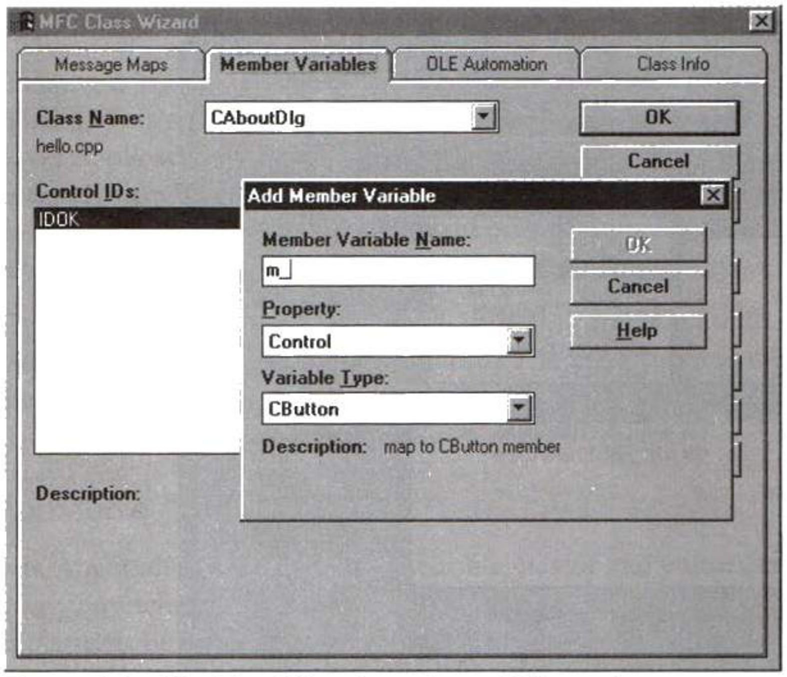
Member Variables

The AppStudio application allows you to include all sorts of controls in your dialog boxes. Edit Input controls can have a variable assigned to them automatically by using the member variables option of the ClassWizard. Figure 4.3 shows an example of the member variables option.

When a variable name is attached to a control, you have the ability to use the variable name instead of issuing messages to the control for certain operations. We will use this option a great deal.

OLE Automation

This option is available for OLE use.



Class Name: CAboutDlg
hello.cpp
Control IDs: IDOK
Description:

Add Member Variable
Member Variable Name: m_
Property: Control
Variable Type: CButton
Description: map to CButton member

Figure 4.3 *The ClassWizard member variables option.*

Class Info

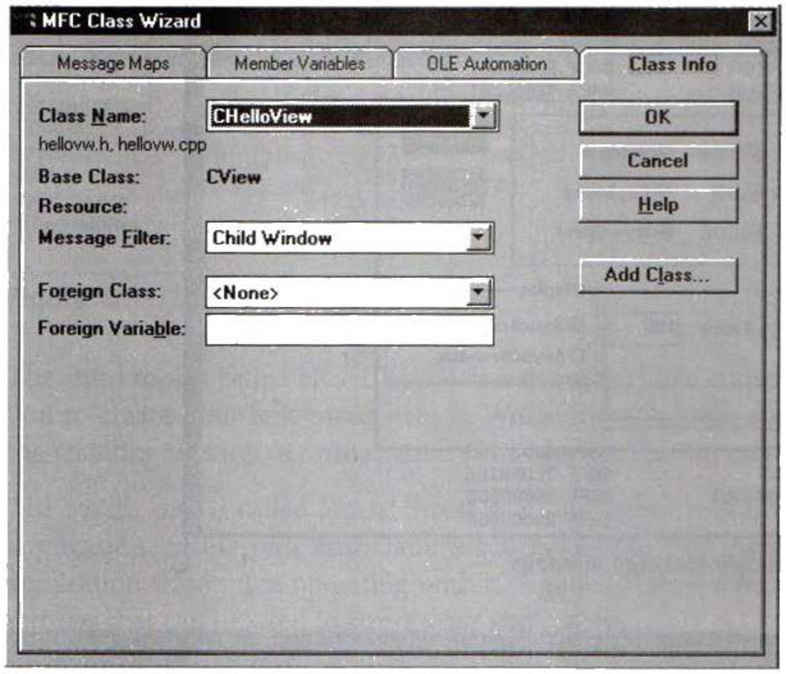
The last option is called Class Info. Figure 4.4 shows how a class looks to the option. You can use the Class Info option to add new classes to your application as well.

Debugger

One of the most important options for a developer is the debugger. Visual C++ includes codeview for Windows. The debugger will allow you to set breakpoints, step through, and monitor the execution of your application.

Source Browser

The Source Browser is used when your application has finished being compiled. It will give you information about the classes and their hierarchy.



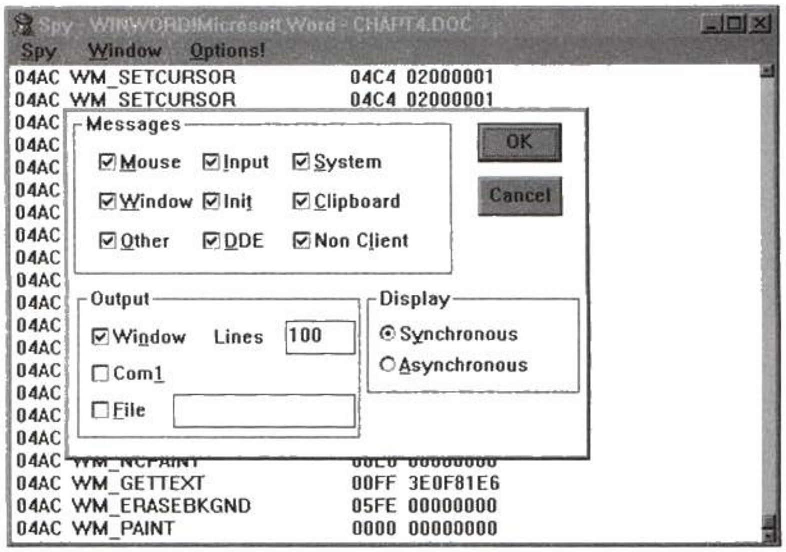
Class Name: CHelloView
hellovw.h, hellovw.cpp
Base Class: CView
Resource:
Message Filter: Child Window
Foreign Class: <None>
Foreign Variable:

Figure 4.4 *The ClassWizard class info option.*

Diagnostic Tools

As mentioned above, the system includes several diagnostic tools that will help the professional developer during application development. Four basic tools are available. The first is called Spy. Spy is an application that executes in the background and displays a single window with the messages that Windows is sending to different applications. An example of the window is shown in Figure 4.5, in which you see a number of different messages being sent to other applications. The dialog box in the middle of the Spy window shows all of the different options available in Spy.

The second tool is called HeapWalker. HeapWalker allows you to look into your memory. Figure 4.6 shows an example of HeapWalker looking at occupied memory. The HeapWalker tool gives you complete information about your memory. You have options available that will give you a list of all free memory as well as a list of Least Recently Used blocks.



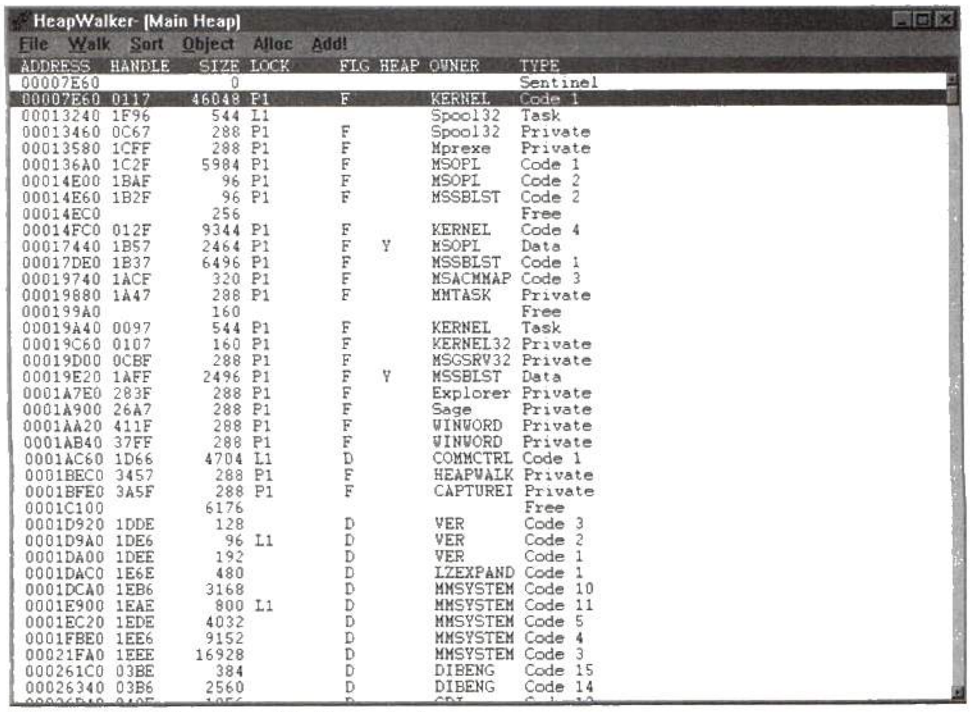
```
04AC WM_SETCURSOR          04C4 02000001
04AC WM_SETCURSOR          04C4 02000001
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC
04AC WM_GETTEXT            00FF 3E0F81E6
04AC WM_ERASEBKGND         05FE 00000000
04AC WM_PAINT              0000 00000000
```

Messages: ☑ Mouse ☑ Input ☑ System ☑ Window ☑ Init ☑ Clipboard ☑ Other ☑ DDE ☑ Non Client

Output: ☑ Window Lines 100 ☐ Com1 ☐ File

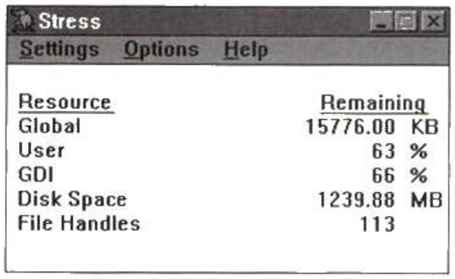
Display: [✓ Synchronous] Asynchronous

Figure 4.5 *The Spy message window.*



```
ADDRESS  HANDLE   SIZE LOCK    FLG HEAP OWNER    TYPE
00007E60             0                           Sentinel
00007E60 0117    46048 P1      F        KERNEL   Code 1
00013240 1F96      544 L1               Spool32  Task
00013460 0C67      288 P1      F        Spool32  Private
00013580 1CFF      288 P1      F        Mprexe   Private
000136A0 1C2F     5984 P1      F        MSOPL    Code 1
00014E00 1BAF       96 P1      F        MSOPL    Code 2
00014E60 1B2F       96 P1      F        MSSBLST  Code 2
00014EC0           256                           Free
00014FC0 012F     9344 P1      F        KERNEL   Code 4
00017440 1B57     2464 P1      F   Y    MSOPL    Data
00017DE0 1B37     6496 P1      F        MSSBLST  Code 1
00019740 1ACF      320 P1      F        MSACMMAP Code 3
00019880 1A47      288 P1      F        MMTASK   Private
000199A0           160                           Free
00019A40 0097      544 P1      F        KERNEL   Task
00019C60 0107      160 P1      F        KERNEL32 Private
00019D00 0CBF      288 P1      F        MSGSRV32 Private
00019E20 1AFF     2496 P1      F   Y    MSSBLST  Data
0001A7E0 283F      288 P1      F        Explorer Private
0001A900 26A7      288 P1      F        Sage     Private
0001AA20 411F      288 P1      F        WINWORD  Private
0001AB40 37FF      288 P1      F        WINWORD  Private
0001AC60 1D66     4704 L1      D        COMMCTRL Code 1
0001BEC0 3457      288 P1      F        HEAPWALK Private
0001BFE0 3A5F      288 P1      F        CAPTUREI Private
0001C100          6176                           Free
0001D920 1DDE      128         D        VER      Code 3
0001D9A0 1DE6       96 L1      D        VER      Code 2
0001DA00 1DEE      192         D        VER      Code 1
0001DAC0 1E6E      480         D        LZEXPAND Code 1
0001DCA0 1EB6     3168         D        MMSYSTEM Code 10
0001E900 1EAE      800 L1      D        MMSYSTEM Code 11
0001EC20 1EDE     4032         D        MMSYSTEM Code 5
0001FBE0 1EE6     9152         D        MMSYSTEM Code 4
00021FA0 1EEE    16928         D        MMSYSTEM Code 3
000261C0 03BE      384         D        DIBENG   Code 15
00026340 03B6     2560         D        DIBENG   Code 14
```

Figure 4.6 *The HeapWalker window.*



```
Resource          Remaining
Global          15776.00 KB
User                  63 %
GDI                   66 %
Disk Space       1239.88 MB
File Handles         113
```

Figure 4.7 *The Stress window.*

The third tool is called HC31. HC31 is a command-line compiler that allows you to create your link-based help in World for Windows and then compile the resulting text into a format your Windows application can use.

The fourth tool is called Stress. Stress is an important tool for stressing your application. Stress will artificially limit the amount of memory that your application thinks it is operating within. Figure 4.7 shows an example of the settings that can be used to stress your application.

Using various options in the Stress application, you can set each of the values to a specific amount. As your application executes, you can monitor the different values to see how your application affects the system and how the settings affect your application.

Help

The Visual C++ system does not include paper documentation. All of the information you need is included on the software's CD-ROM. Most of this documentation is in the Help system. Complete information including examples is provided for the C/C++ languages, the Microsoft Foundation Classes, and also the Windows 3.1 Software Development Kit. There is no doubt you will be able to find the information you need.

Microsoft Foundation Classes

At the heart of the Visual C++ system are the Microsoft Foundation Classes. The entire class structure includes over 100 classes. The classes are broken

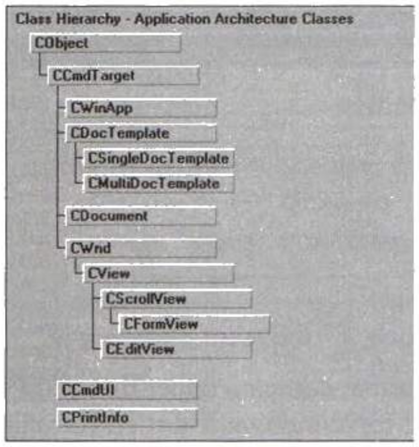


Figure 4.8 *The Application architecture hierarchy.*

down into five areas: Application Architecture, Visual Object, General Purpose, OLE2, and Database.

Application Architecture

Figure 4.8 shows the object hierarchy for the application architecture classes. The classes that make up the application architecture hierarchy include those classes that are used in the creation of a Windows Application. There are three categories of classes: Windows Application, Command-Related, and Document/View.

The Windows Application class includes the CWinApp class. This is the fundamental class for all applications. It includes code for initializing, running, and terminating a Windows program. The Command-Related class contains the base classes for objects that are able to send and receive messages. The Document/View class provides the base classes for the creation and viewing of windows.

Visual Object

Figure 4.9 shows the object hierarchy for the visual object classes. The classes that make up the visual object hierarchy include those classes that are used in the view of a window. The categories of classes include window, view, dialog,

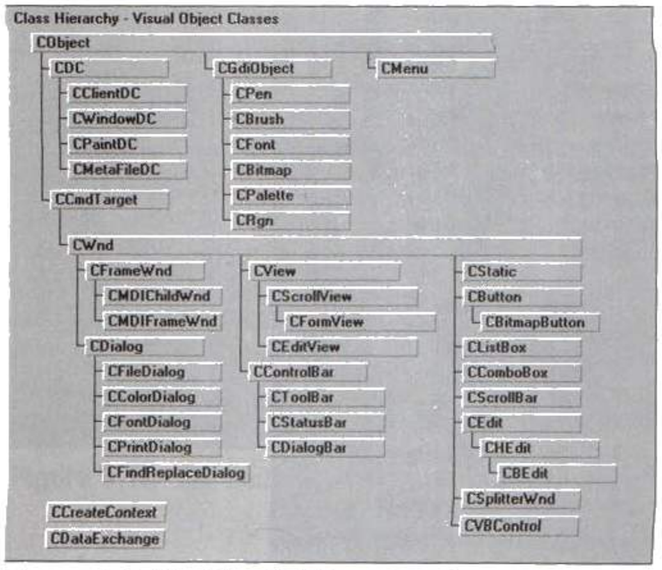


Figure 4.9 *The Visual Object hierarchy.*

control, menu, device context, and drawing object. Clearly, all of the classes deal with the actual windows and the mechanisms for drawing on the windows.

General Purpose

Figure 4.10 shows the object hierarchy for the General Purpose classes. The classes that make up the General Purpose hierarchy include those classes that are used by other classes. The categories are root, file, diagnostic, exception, collections, and miscellaneous support classes. The root category includes the Cobject class, which is the base class of nearly all classes in the Microsoft Foundation Classes library.

OLE2

Figure 4.11 shows the object hierarchy for the OLE2 classes. The classes that make up the OLE2 hierarchy include those classes that are used in applications that use OLE2. The categories include OLE base, OLE container, OLE data transfer, OLE server, OLE dialog box, and OLE miscellaneous.

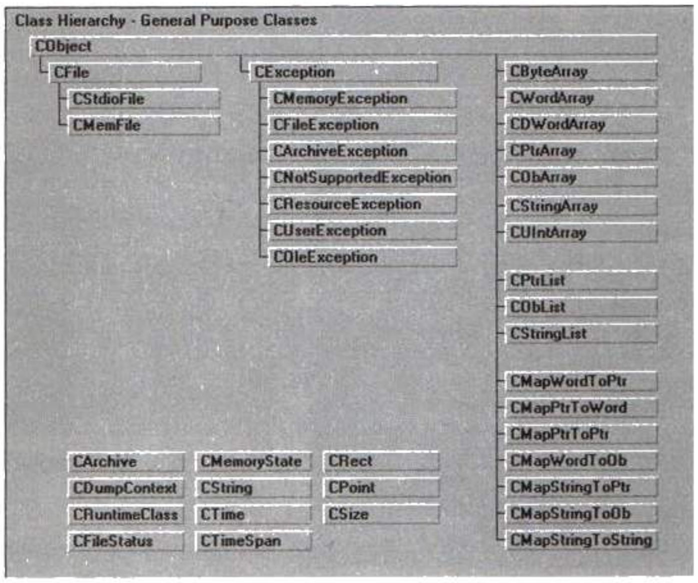


Figure 4.10 *The General Purpose hierarchy.*

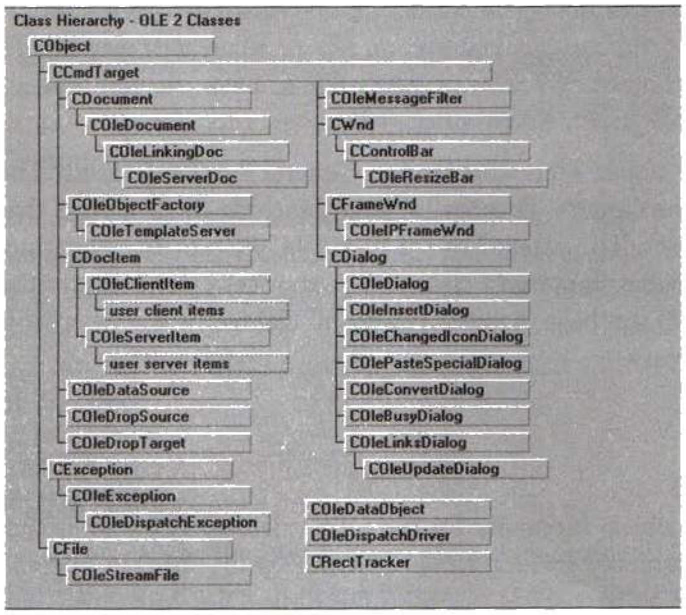


Figure 4.11 *The OLE2 hierarchy.*

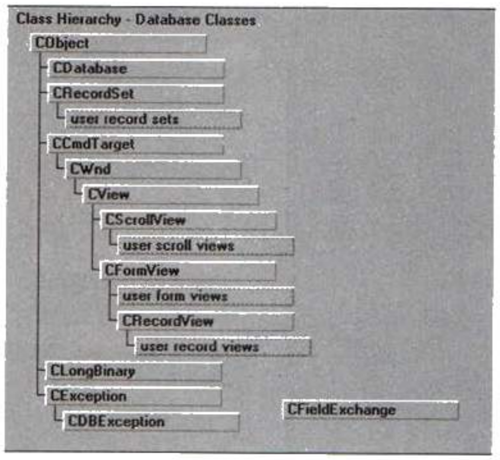


Figure 4.12 *The Database hierarchy.*

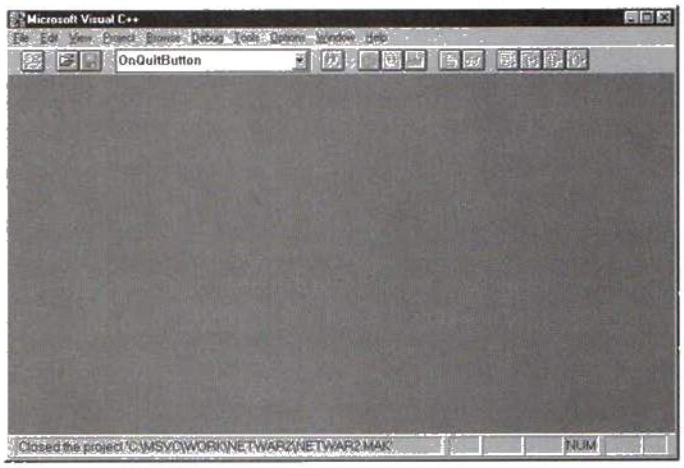
Database

Figure 4.12 shows the object hierarchy for the database classes. The classes that make up the Database hierarchy include those classes that are used to create database applications.

Programming

Now that we know what we are dealing with, it's time to create a couple of programs. We will follow the programs done in Chapter 2 in order to see the differences in programming style. Because this book is intended to be a tutorial on designing and developing Windows network games, we will be skipping many of the details and features of the Visual C++ system. You would be advised to pick up a book specifically written to expound the Visual C++ system.

One case in point is our example programs which deal with SDI or Single Document Interface applications. This means that we will only have one main window. An MDI or Multiple Document Interface application has several main windows. As an example, the notepad accessory that is included with Microsoft Windows is an SDI application, whereas Visual C++ is an MDI application.


Microsoft Visual C++
File Edit View Project Browse Debug Tools Options Window Help
OnQuitButton
NUM

Closed the project 'C:\MSVC\WORK\NETWAR2\NETWAR2.MAK'

Figure 4.13 *Startup screen.*

"Hello World"

Let's get started. We are going to use the AppWizard to generate the skeleton for our "Hello World" program. Start by launching Visual C++. You should get a screen that looks like Figure 4.13.

Step 1: Execute the AppWizard to create our skeleton SDI application.

The AppWizard is located under the Project Menu heading. Click on Project, then click on AppWizard. You will see the MFC AppWizard dialog box (Figure 4.14). This is the first of several dialog boxes that you will encounter as the application is created.

Your dialog box will look like the one above but will not have any text in the white parts. Click on the edit line for the Project Name and enter the name "hello". Your dialog box should now look just like the one above. You will notice that the AppWizard names a subdirectory for the project automatically. Now before you click on the OK button, we have to customize our application a little, so click on the Options button.

A new dialog box will appear. Click on the appropriate options until your Options dialog box looks like the one in Figure 4.15. Click on OK to remove the Options dialog box. When the AppWizard dialog box appears again, click on its OK button. Just as soon as you click the OK button, you will be

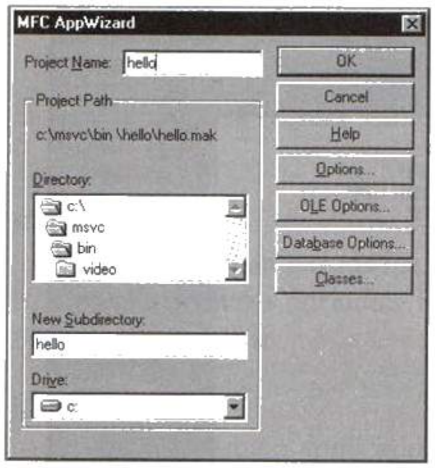
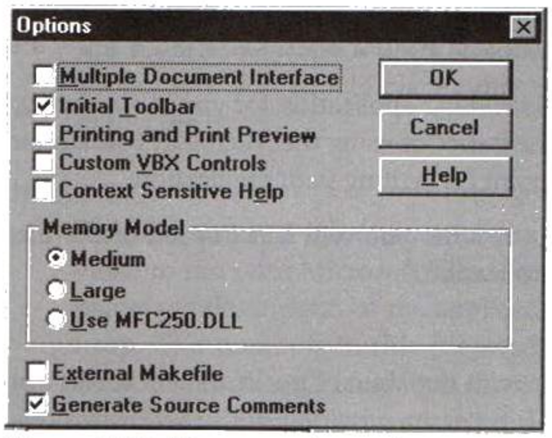


Figure 4.14 *The MFC AppWizard dialog box.*

presented with a New Application Information dialog box like the one shown in Figure 4.16.

This dialog box gives you information about the application you are creating. Make sure that the system is creating an SDI application. Click the Create button to have AppWizard create your application. You will see the files created in a small dialog box on the screen. After the files for your application are created, you will see a blank window. The files created by AppWizard are

```
hello.mak, hello.def, hellovw.cpp, hello.rc, hellovw.h, resource.h, etc.
```

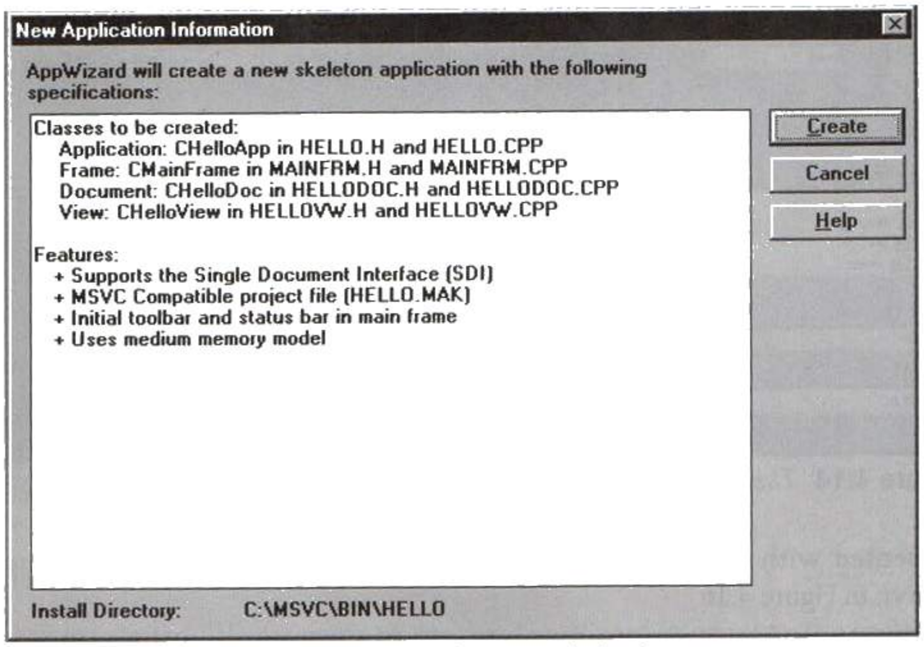

Options

- [ ] Multiple Document Interface
- [✓] Initial Toolbar
- [ ] Printing and Print Preview
- [ ] Custom VBX Controls
- [ ] Context Sensitive Help

Memory Model
- [✓ Medium]
- Large
- Use MFC250.DLL

- [ ] External Makefile
- [✓] Generate Source Comments

Figure 4.15 *The appropriate options.*



AppWizard will create a new skeleton application with the following specifications:

```
Classes to be created:
    Application: CHelloApp in HELLO.H and HELLO.CPP
    Frame: CMainFrame in MAINFRM.H and MAINFRM.CPP
    Document: CHelloDoc in HELLODOC.H and HELLODOC.CPP
    View: CHelloView in HELLOVW.H and HELLOVW.CPP

Features:
   + Supports the Single Document Interface (SDI)
   + MSVC Compatible project file (HELLO.MAK)
   + Initial toolbar and status bar in main frame
   + Uses medium memory model
```

Install Directory: C:\MSVC\BIN\HELLO

Figure 4.16 *The New Application Information dialog box.*

The files created are described in the file readme.txt. This file says the following about our application:

```
========================================
        MICROSOFT FOUNDATION CLASS LIBRARY : HELLO
========================================
```

AppWizard has generated this HELLO application for you. This application not only demonstrates the basics of using the Microsoft Foundation classes, but is also a starting point for writing your application.

This file contains a summary of what you will find in each of the files that make up your HELLO application.

HELLO.MAK

This project file is compatible with the Visual C++ Workbench. It is also compatible with the NMAKE program provided with the Professional

Edition of Visual C++. To build a debug version of the program from the MS-DOS prompt, type `nmake DEBUG=1 /f HELLO.MAK`

or to build a release version of the program, type `nmake DEBUG=0 /f HELLO.MAK`

HELLO.H

This is the main include file for the application. It includes other project specific includes (including RESOURCE.H) and declares the CHelloApp application class.

HELLO.CPP

This is the main application source file that contains the application class CHelloApp.

HELLO.RC

This is a listing of all of the Microsoft Windows resources that the program uses. It includes the icons, bitmaps, and cursors that are stored in the RES subdirectory. This file can be directly edited with App Studio.

RES\HELLO.ICO

This is an icon file, which is used as the application's icon. This icon is included by the main resource file HELLO.RC.

RES\HELLO.RC2

This file contains resources that are not edited by App Studio. Initially, this contains a VERSIONINFO resource that you can customize for your application. You should place other non-App Studio editable resources in this file.

HELLO.DEF

This file contains information about the application that must be provided to run with Microsoft Windows. It defines parameters such as the name and description of the application, and the size of the initial local heap. The numbers in this file are typical for applications developed with the Microsoft Foundation Class Library. The default stack size can be adjusted by editing the project file.

HELLO.CLW

This file contains information used by ClassWizard to edit existing classes or add new classes. ClassWizard also uses this file to store information needed to generate and edit message maps and dialog data maps and to generate prototype member functions.

///

For the main frame window:

MAINFRM.H, MAINFRM.CPP

These files contain the frame class CMainFrame, which is derived from CFrameWnd and controls all SDI frame features.

RES\TOOLBAR.BMP

This bitmap file is used to create tiled images for the toolbar. The initial toolbar and status bar are constructed in the CMainFrame class. Edit this toolbar bitmap along with the array in MAINFRM.CPP to add more toolbar buttons.

///

AppWizard creates one document type and one view:

HELLODOC.H, HELLODOC.CPP - the document

These files contain your CHelloDoc class. Edit these files to add your special document data and to implement file saving and loading (via CHelloDoc::Serialize).

HELLOVW.H, HELLOVW.CPP - the view of the document

These files contain your CHelloView class.

CHelloView objects are used to view CHelloDoc objects.

///

Other standard files:

STDAFX.H, STDAFX.CPP

These files are used to build a precompiled header (PCH) file named

STDAFX.PCH and a precompiled types (PCT) file named STDAFX.OBJ.

RESOURCE.H

This is the standard header file, which defines new resource IDs. App Studio reads and updates this file.

///

Other notes:

AppWizard uses "TODO:" to indicate parts of the source code you should add to or customize.

///

As you can see, the AppWizard does a thorough job of creating the application for us. In fact, all we have to do now is compile the code.

Step 2: Compile the code AppWizard has created.

All of the compiling for our application will be controlled by a project file. The project file called *hello.mak* keeps track of all the files in our application as well as all of the dependencies between the files. To compile our application, click on the Project menu item. You will see several options in the pop-down menu. These options include:

Build hello.exe

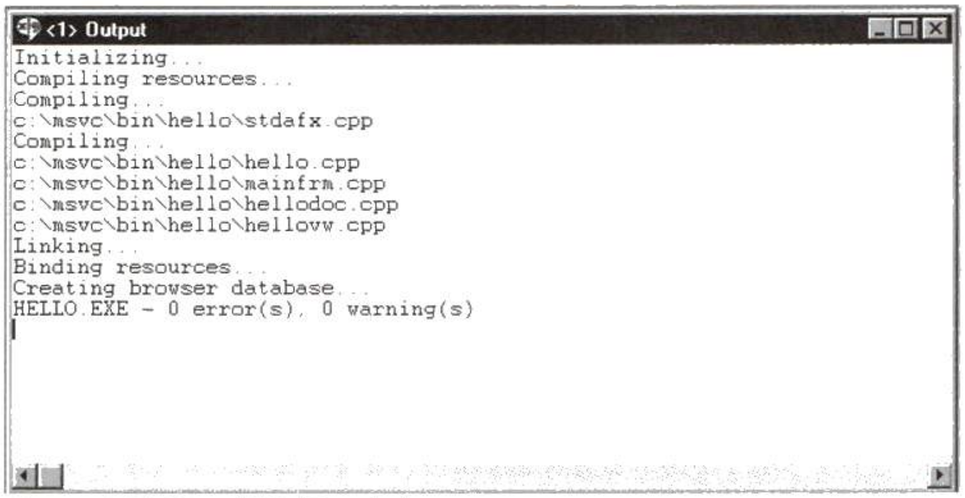
Rebuild All hello.exe

Execute hello.exe

Click on the option for *Build hello.exe.* If everything compiles successfully, as it should, you will see the window in the Visual WorkBench client region as shown in Figure 4.17. Now it's time to execute the application.

Step 3: Execution

To execute the application, click on the Project menu item again and click on *Execute Hello.exe.* You should get a window like Figure 4.18. The application has a menu bar and even a toolbar. Click on File and exit to terminate the application.



```
Initializing...
Compiling resources...
Compiling...
c:\msvc\bin\hello\stdafx.cpp
Compiling...
c:\msvc\bin\hello\hello.cpp
c:\msvc\bin\hello\mainfrm.cpp
c:\msvc\bin\hello\hellodoc.cpp
c:\msvc\bin\hello\hellovw.cpp
Linking...
Binding resources...
Creating browser database...
HELLO.EXE - 0 error(s), 0 warning(s)
```

Figure 4.17 *Visual WorkBench client region.*

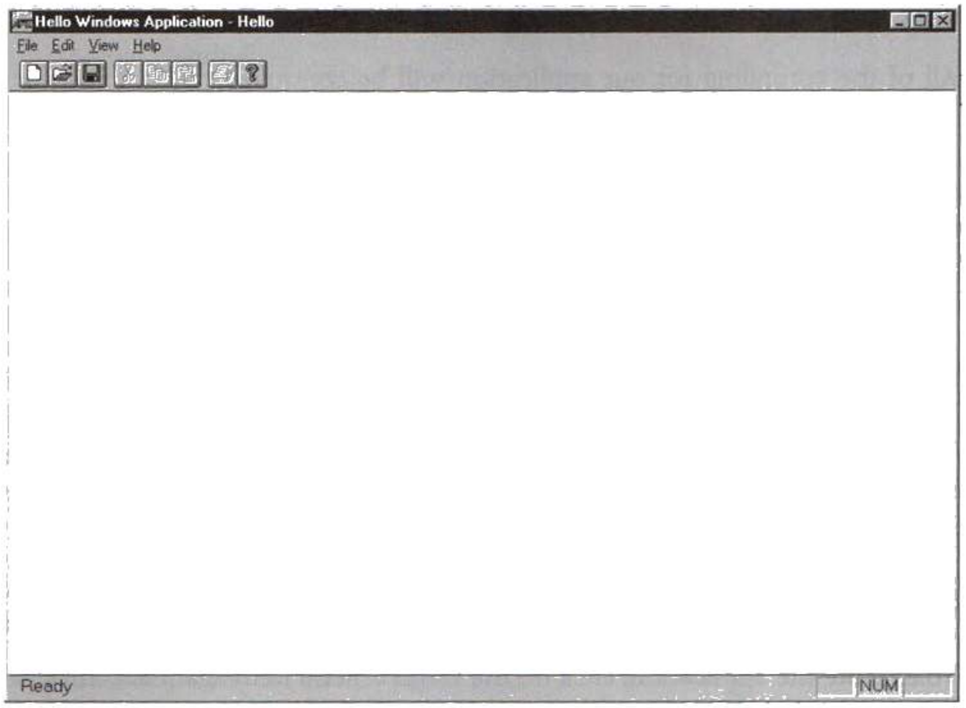


Figure 4.18 *Executing an application.*

Adding Graphics

The application looks good and works well, but there isn't any "Hello World" anywhere in the window. Recall from Chapter 3 that when we were drawing in the window, we said that the best place to draw was in the *case* statement for the WM_PAINT message. Let's just do the same thing here. Using the file open toolbar button, open the hellovw.cpp file. The file should be opened and displayed in the client region of the Visual C++ WorkBench. Move through the code and try to find our message handler function. It's not there. Visual C++ and MFC handle things just a bit differently than C and the SDK.

When the AppWizard created our "Hello World" program, it created several classes based on classes in the MFC. One of the classes is called CHelloView. This is the class for our window and is called the View class. One of the member functions of the View classes is called OnDraw. This is a virtual member function that is called each time the window needs to be repainted. This should sound familiar. We know that the Windows system sent our application the WM_PAINT message each time the window needed repainting. This function acts like the *case* statement we used in the C Windows programs in Chapter 3.

We also know from the previous chapter that Windows does not allow us to draw directly to the screen. We have to use a device context. In Visual C++, the device context is an object of class CDC. Conveniently, an object of class CDC is passed to the OnDraw() member function automatically. This makes sense because the OnDraw() function is used to repaint the window and you have to use a device context to do the repainting.

So, let's add some code to our program. Look back at the code displayed on your WorkBench. Move through it until you find the code:

```
//////////////////////////////////////////////////////////////////////////////
//CHelloView drawing
void CHelloView::OnDraw(CDC* pDC)
{
      CHelloDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);
      // TODO: add draw code for native data here
}
```

Replace all of the text within the two brackets so you have the following:

```
void CHelloView::OnDraw(CDC* pDC)
```

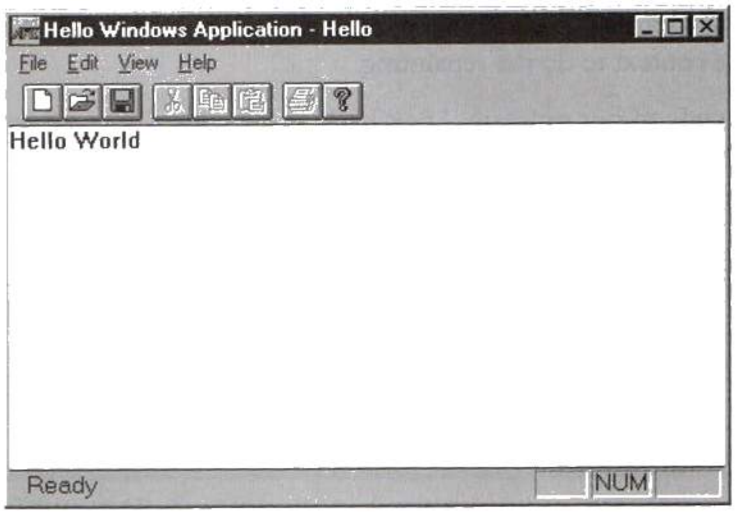
```
{
     pDC->TextOut(0, 0, "Hello World");
}
```

Now recompile and execute the program again. You should have the screen shown in Figure 4.19. Just one line of code is all that it takes to draw our text on the screen.

App Studio

A very important part of the Visual C++ system is the App Studio. If you have ever done extensive Windows programming using the SDK and C, you know what a pain it can be to create your menus and dialog boxes. Everything must be done by hand. App Studio was designed to not only provide a simple "paint-like" program for adding menus, accelerators, and dialog boxes to your application, but to give a seamless option where your dialog boxes and other components are automatically integrated into the application.

Throughout this book, we will be using the App Studio. I would suggest reading up on it. The App Studio application references the .rc file created by the App-Wizard. This file, called the resource file, includes all of the information about dialog boxes and their associated controls as well as menus, icons, and bitmaps.



Hello World

Figure 4.19 *Recompile and execute again.*

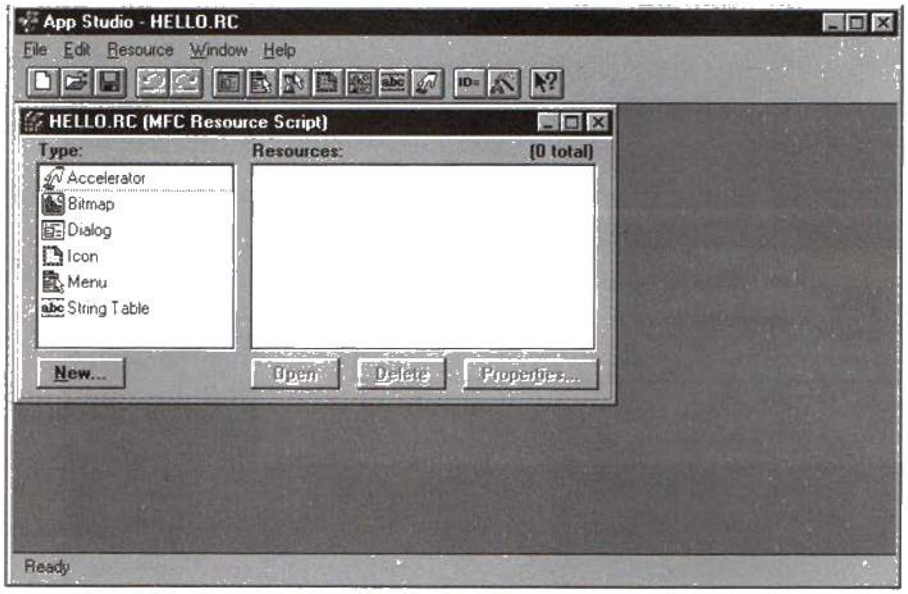


Figure 4.20 *How to execute App Studio.*

App Studio is executed by clicking on the Tools menu item and then clicking on App Studio. Do this now. You will be shown the window seen in Figure 4.20.

Inside the App Studio window is a child window with the name of the resource file that is being edited. Your window should have the text "hello.rc" on it. The left part of the window lists all of the different resource types available to you. When one of the types is highlighted, all of the resources of this type will be listed by name on the right part of the window. To select a resource just double-click on its name.

In the case of our "Hello World" program, the AppWizard created at least one of every type of resource. As a simple example of using the App Studio, let's make a change to our application's About dialog box.

Click on the Dialog resource type value. A resource called IDD_ABOUTBOX will appear in the list of resources. Double-click on this resource to bring up the dialog box and editing screen as shown in Figure 4.21. Make a few changes to the dialog box. An example of several changes is shown in Figure 4.22.

Now click on the disk icon in the toolbar and close the App Studio application. To see the changes to the About dialog box, you will need to recompile

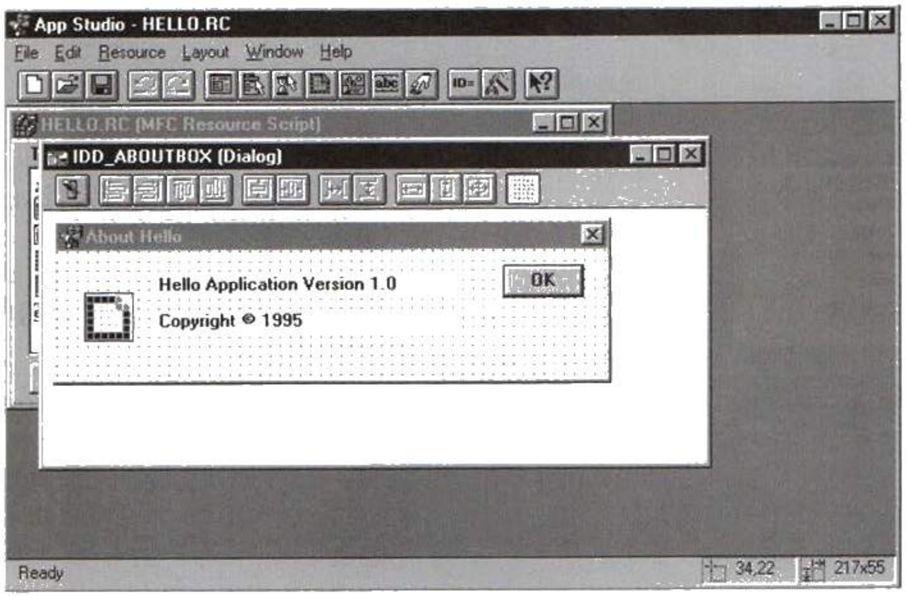

App Studio - HELLO.RC
File Edit Resource Layout Window Help
HELLO.RC (MFC Resource Script)
IDD_ABOUTBOX (Dialog)
About Hello
Hello Application Version 1.0
Copyright © 1995
OK
Ready
34,22
217x55

Figure 4.21 *Double-clicking on IDD_ABOUTBOX.*

the "Hello World" project. Click on Project, then click on Build hello.exe. After several seconds, you will be able to execute the application. Click on the Help menu item and then click on About.

Using the Mouse

Our "Hello World" program was pretty simple, but a good first program for using Visual C++. Now we can expand things a bit and see how to add mouse actions to the program. Recall from Chapter 3 that we added the

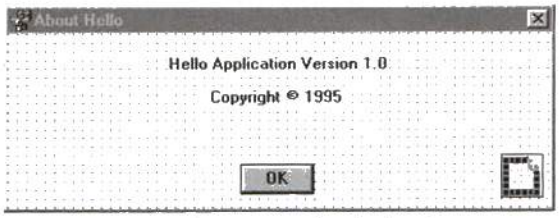
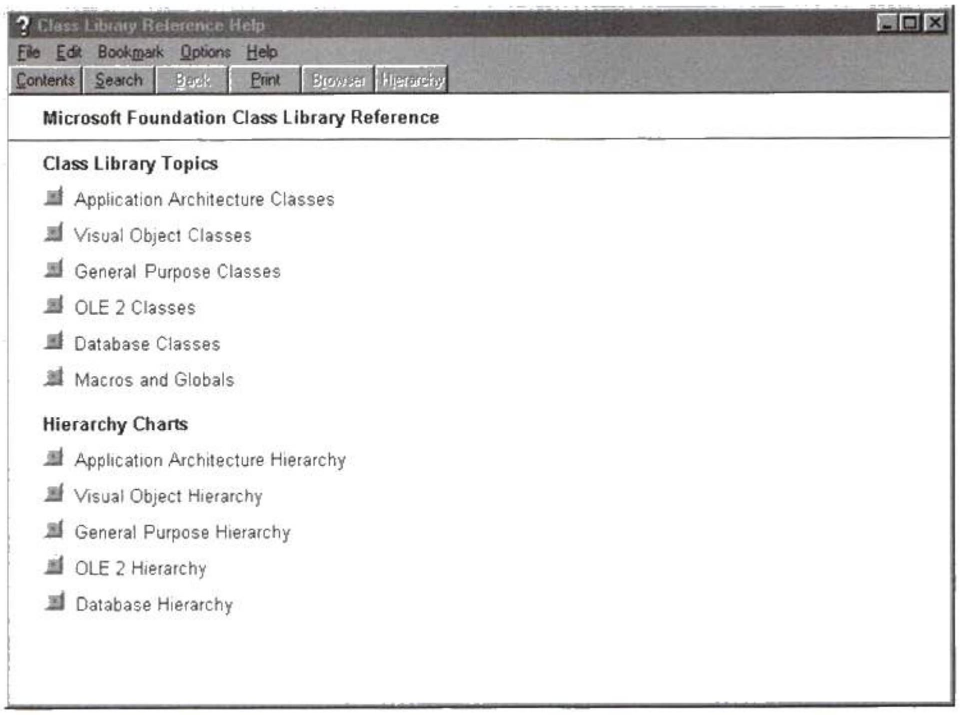


Figure 4.22 *Changes to the dialog box.*


Class Library Reference Help
File Edit Bookmark Options Help
Contents Search Back Print Browser Hierarchy

Microsoft Foundation Class Library Reference

Class Library Topics

- Application Architecture Classes
- Visual Object Classes
- General Purpose Classes
- OLE 2 Classes
- Database Classes
- Macros and Globals

Hierarchy Charts

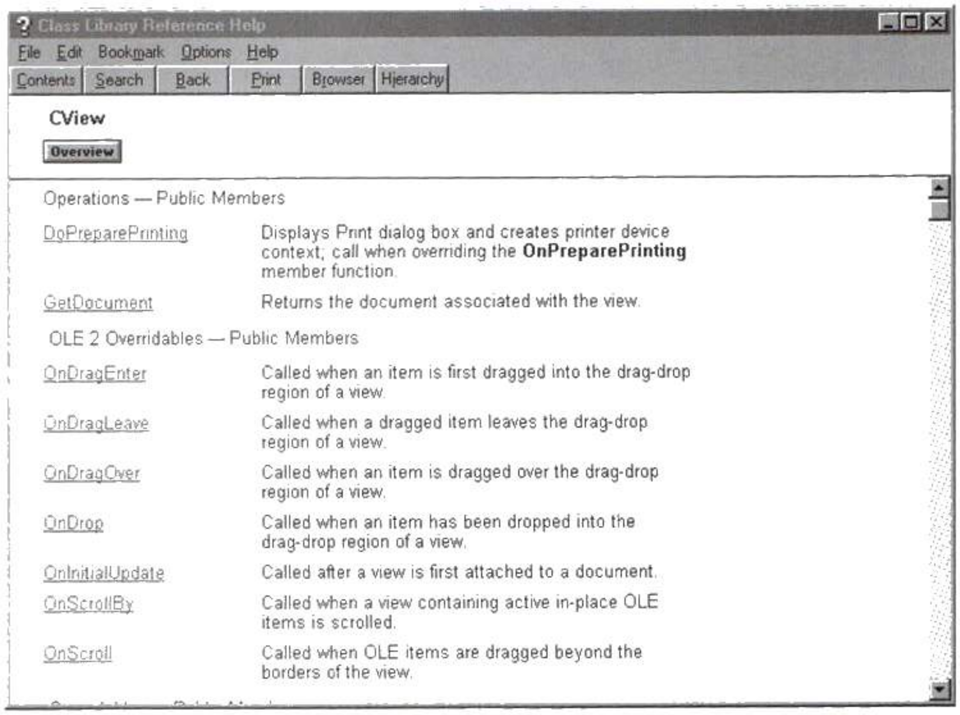
- Application Architecture Hierarchy
- Visual Object Hierarchy
- General Purpose Hierarchy
- OLE 2 Hierarchy
- Database Hierarchy

Figure 4.23 *Foundation classes.*

mouse message *case* statements so that the location of the mouse would be put on the screen. You will also recall that earlier in this chapter, I sent you looking for the *switch* statement and associated control function that handles all of the messages that come from the system. You were not able to find the function because Visual C++ handles things differently.

To get an idea about how things work in Visual C++, you need to look at the documentation for the CView class. Remember that our window is a class called CHelloView that is derived from CView. In Visual C++, click on Help and then Foundation Classes. You will see the screen shown in Figure 4.23. Click on Visual Object Classes and then click on the entry for CView. You will see the screen shown in Figure 4.24.

This screen shows all of the member functions for the CView class. The CView class is derived from Cwnd, and together, the two classes contain many hundreds of functions. If you look through the list, you will find mem-


Class Library Reference Help
File Edit Bookmark Options Help
Contents Search Back Print Browser Hierarchy
Overview

CView

Operations — Public Members

DoPreparePrinting	Displays Print dialog box and creates printer device context; call when overriding the **OnPreparePrinting** member function.
GetDocument	Returns the document associated with the view.

OLE 2 Overridables — Public Members

OnDragEnter	Called when an item is first dragged into the drag-drop region of a view.
OnDragLeave	Called when a dragged item leaves the drag-drop region of a view.
OnDragOver	Called when an item is dragged over the drag-drop region of a view.
OnDrop	Called when an item has been dropped into the drag-drop region of a view.
OnInitialUpdate	Called after a view is first attached to a document.
OnScrollBy	Called when a view containing active in-place OLE items is scrolled.
OnScroll	Called when OLE items are dragged beyond the borders of the view.

Figure 4.24 *The member functions for the CView class.*

ber functions that start with the characters "On". These member functions are responses to various Windows events. They are directly related to the messages that Windows sends to the application when, say, a key is pressed. We have to use these functions to respond to the mouse as well.

Handling Messages

Messages in Visual C++ are handled using macros and the member functions mentioned earlier. If you want to respond to a click of the left mouse button, you will need to have a member function in your view class such as:

```
void CHelloView::OnLButtonDown(UINT flags, Cpoint point)
{
    // do something here
}
```

In the header file (.h) for your class (hellovw.h), you will have to have a prototype of the member function such as:

```
afx_msg void OnLButtonDown(UINT flags, Cpoint point)
```

You will also have to have a message map macro to connect the message to the member function such as:

```
BEGIN_MESSAGE_MAP(CHelloView, Cview)
   ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

These steps will allow a left mouse click to trigger a WM_LBUTTON-DOWN message that in turn causes the execution of the OnLButton-Down() function.

ClassWizard

You might be saying to yourself that this doesn't seem like much help. But wait a minute, Microsoft has included another application in the Visual C++ system to help with all of this. The application is called the ClassWizard.

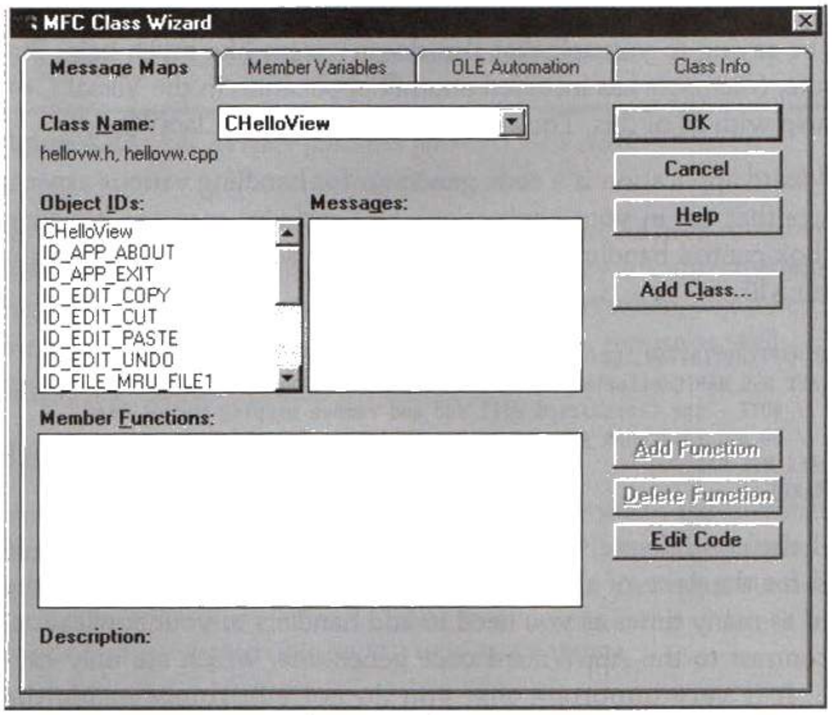
The ClassWizard application is a code generator for handling various aspects of the objects that are in your application. This includes message handling and dialog box control handling. If you look at your source code, you might find something like:

```
BEGIN_MESSAGE_MAP(CHelloView, CView)
        //{{AFX_MSG_MAP(CHelloView)
            // NOTE - the ClassWizard will add and remove mapping macros here.
            // DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The part of the code above that starts *//{{AFX_MSG_MAP(CHelloView)* is code added for the sake of the ClassWizard. You are able to execute the ClassWizard as many times as you need to add handlers to your application. This is in contrast to the AppWizard code generators, which are only executed once. It is very important that you do not edit things inside the AFX_MSG statements.

So, let's execute ClassWizard and handle the mouse. To execute ClassWizard, click on Browse, then on ClassWizard. You will see the window shown in Figure 4.25. The ClassWizard has four areas that you can use: Message Maps, Member Variables, OLE Automation, and Class Info. We will be using the first two only. We are going to respond to the right and left mouse buttons. We want the mouse messages to be sent to our view object. Make sure that the class CHelloView in visible in the Class Name entry. If it is not, click on the selection arrow to find the correct class. When you are using the Message Map option, it is very important that you make sure the correct class is being used.

Once the CHelloView class is visible, you will see a number of entries under the left-side heading, Object IDs. The IDs that start out "ID_" are all related to the menu of our application and can be ignored for this exercise. The very first entry is the one we are interested in. It should read CHelloView. Click on this



Class Name: CHelloView
hellovw.h, hellovw.cpp

Object IDs:
CHelloView
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_COPY
ID_EDIT_CUT
ID_EDIT_PASTE
ID_EDIT_UNDO
ID_FILE_MRU_FILE1

Messages:

Member Functions:

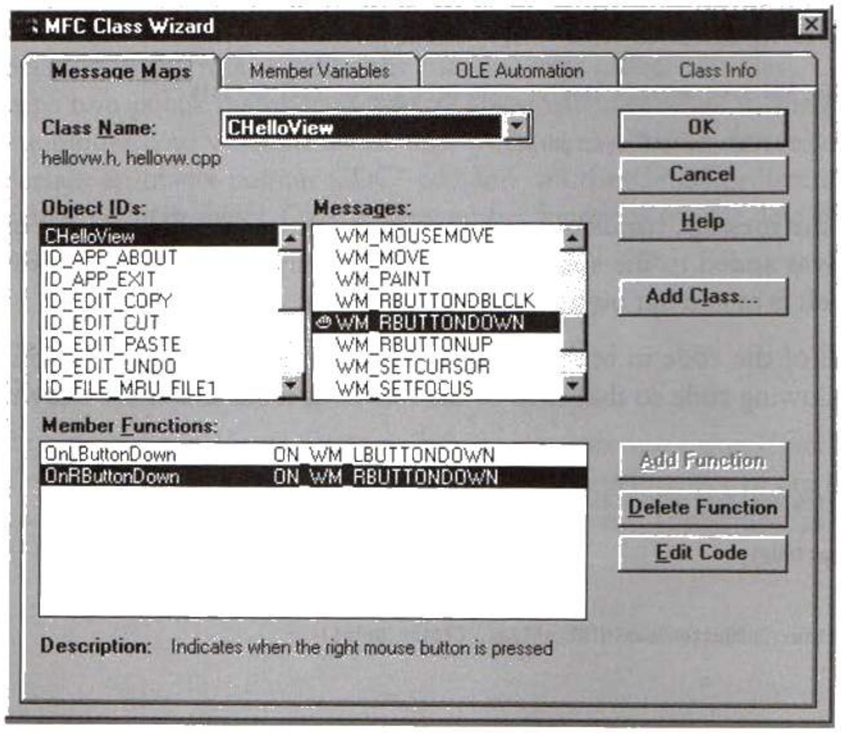
Description:

Figure 4.25 *Executing ClassWizard.*

entry to highlight it. Once you do this, a list of options will appear under the right-side heading Messages. This list is all of the messages for our View class that we are allowed to respond to. These are basically system messages.

Move down and find the entry for WM_LBUTTONDOWN and click on it. You will see a description of the message at the bottom of the ClassWizard window. We want to respond to this message, so click on the Add Function button. The function will appear at the bottom of the ClassWizard window under the heading Member Functions. Now find WM_RBUTTONDOWN and do the same thing. After you have added both functions, your window should appear as in Figure 4.26.

Now click on OK. The ClassWizard window will disappear, and you will be returned to the WorkBench of Visual C++. To see the changes the ClassWizard



Class Name: CHelloView
hellovw.h, hellovw.cpp

Object IDs:
CHelloView
ID_APP_ABOUT
ID_APP_EXIT
ID_EDIT_COPY
ID_EDIT_CUT
ID_EDIT_PASTE
ID_EDIT_UNDO
ID_FILE_MRU_FILE1

Messages:
WM_MOUSEMOVE
WM_MOVE
WM_PAINT
WM_RBUTTONDBLCLK
WM_RBUTTONDOWN
WM_RBUTTONUP
WM_SETCURSOR
WM_SETFOCUS

Member Functions:

OnLButtonDown	ON_WM_LBUTTONDOWN
OnRButtonDown	ON_WM_RBUTTONDOWN

Description: Indicates when the right mouse button is pressed

Figure 4.26 *Responding to a message in the ClassWizard window.*

made to our source code, open the hellovw.cpp file. At the top of the file you will see the message map code

```
BEGIN_MESSAGE_MAP(CHelloView, CView)
      //{{AFX_MSG_MAP(CHelloView)
      ON_WM_LBUTTONDOWN()
      ON_WM_RBUTTONDOWN()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The code includes two macros for the left and right buttons of the mouse. Now move to the bottom of the source file until you find the code

```
//////////////////////////////////////////////////////////////////////////////
// CHelloView message handlers
void CHelloView::OnLButtonDown(UINT nFlags, CPoint point)
{
      // TODO: Add your message handler code here and/or call default

      CView::OnLButtonDown(nFlags, point);
}
void CHelloView::OnRButtonDown(UINT nFlags, CPoint point)
{
      // TODO: Add your message handler code here and/or call default

      CView::OnRButtonDown(nFlags, point);
}
```

These are the message handlers for the left and right mouse buttons. All of this code was added to the source code automatically by the ClassWizard. All that is left is to add our own code to the functions.

Remove all of the code in both functions between the two brackets. Now add the following code so that each of the functions looks like those shown below.

```
void CHelloView::OnLButtonDown(UINT nFlags, CPoint point)
{
   Invalidate(TRUE);
}

void CHelloView::OnRButtonDown(UINT nFlags, CPoint point)
{
   CDC* cdc;

   cdc = GetDC();
```

```
    cdc->TextOut(point.x, point.y, "Right Button Clicked");
  ReleaseDC(cdc);
}
```

Build the application again and execute it. Click on the right mouse button a couple of times in different locations of the screen. Now click on the left mouse button. This acts just like one of our programs in Chapter 3. Sure is easier to program.

Look back at the code in the function for the left mouse button. The only statement in the function is *Invalidate(TRUE);*. This is the same statement as the SDK statement *InvalidateRect(hwnd, NULL);*. Something you will notice about this statement and others is that we don't have to specify a window handle for the statement to act upon because our functions are member functions of our view object. The statements automatically know which window to use.

In the right mouse button function you will see that we make use of a parameter sent to the function. The second parameter called point is the current position of the mouse when the function was called. The class Cpoint contains two public members X and Y. These member relate to the position of the mouse. You will also notice that we have to obtain a device context. We declare an object pointer CDC* cdc and call the GetDC(); function for the device context object. One of the member functions for the device context is TextOut. This certainly is a clean way of doing things.

Summary

Programming in Visual C++ is a breeze compared to using C and the SDK. You will be amazed as we move through the development of our game. The package itself will keep track of the classes in your application as well as all of the dialog boxes and other controls.

Chapter 5

King's Reign Foundation

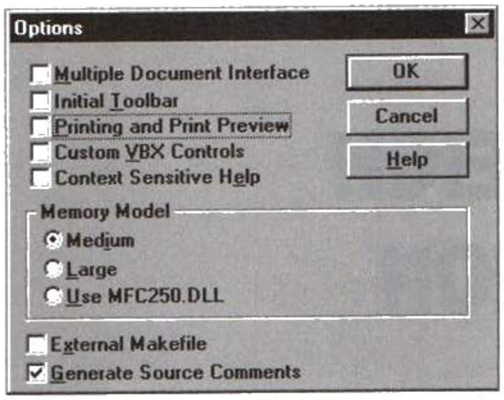
With the introductions out of the way, it's time to start building our game. We will do the construction in a series of steps. The outline is:

- basic foundation
- operations window and functions
- game details
- Internet support
- modem support

By breaking the game development into pieces, we can make sure that all parts work correctly before adding more complexity. At the end of each outline point, you will have a complete game that is completely functional, so let's get started.

Creating the Main Window Class

Since we are using Visual C++ and the MFC library, we might as well let the system do most of the work for us. This includes building the support code for the Windows interface. Begin by executing the Visual C++ system. Use the AppWizard to create a new application using the name NETWAR2. Make sure that you select the Options button and select the appropriate options based on the screen shown in Figure 5.1.


Options

- [ ] Multiple Document Interface
- [ ] Initial Toolbar
- [ ] Printing and Print Preview
- [ ] Custom VBX Controls
- [ ] Context Sensitive Help

Memory Model

- [✓ Medium]
- Large
- Use MFC250.DLL

- [ ] External Makefile
- [x] Generate Source Comments

Figure 5.1 *Selecting the appropriate options.*

After a time, the system will finish creating the necessary files and you will have a complete Windows application. The files for the game should be located in the /netwar2 directory. The AppWizard created our view class when it generated the game. The View class is called CNetwar2View and is defined in the netwavw.h file. You should open this file and get familiar with our View class.

With the application generated, it can be compiled. Select the Project menu item and select the Build netwar2.exe menu item to compile the code. Once compiled, select Project again and then Execute netwar2.exe. You should get a window that looks like Figure 5.2.

When we first discussed creating this book and the associated game, we had to spend some time talking about the type of system to develop the game for. In the end, we decided that programming the game for an 80486 machine with a minimum of a 640 × 480 screen would encompass the most people. For this reason, we have to make a few changes to the base code so that the game's window fits in a 640 × 480 screen.

Screen Size

In the MFC libraries, all of the classes build upon others. Some of the classes have methods that can be overridden with our own methods. To change the screen size and make sure that it is the same size each time the game is exe-

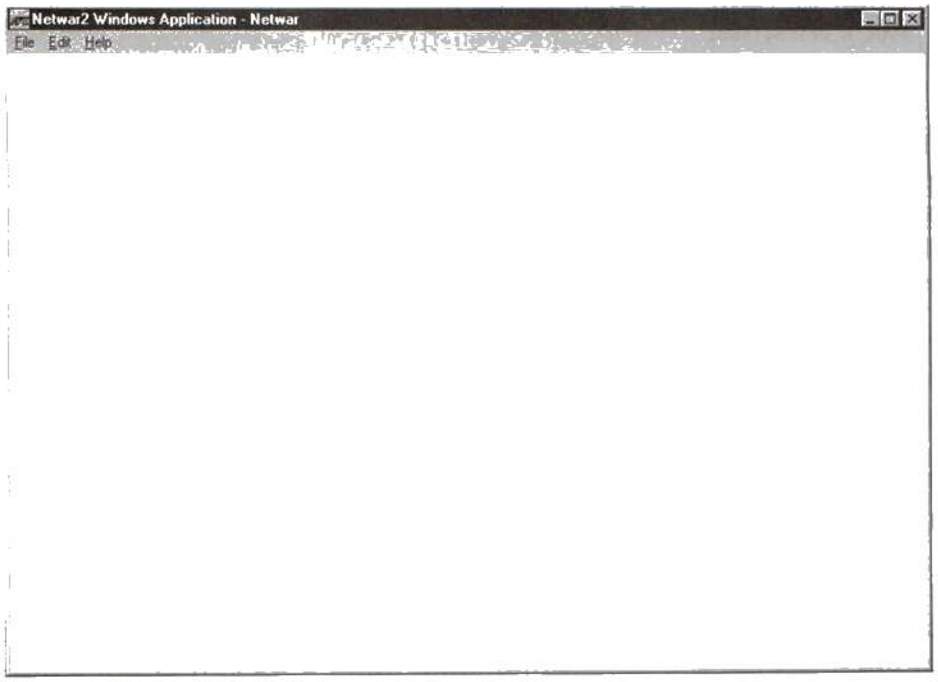


Figure 5.2 *Compiling the generated application.*

cuted, we have to override a specific method of the CFrameWnd class. The CFrameWnd class is a parent class for our window. The method that we are overriding is called ActivateFrame().

To make the changes, begin by opening the file mainfrm.h. This is the header file for the class that creates our window. At the beginning of the file will be the text

```
class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
      CMainFrame();
      DECLARE_DYNCREATE(CMainFrame)

and more...
```

Insert the text

```
private:
    virtual void ActivateFrame(int nCmdShow = -1);
    BOOL m_bFirstTime;
```

between the { and the word *protected:*.

The word *private* means that this information is private to this class and cannot be accessed by any other class. The next line is the declaration of the function we are overriding. The third line is a declaration of a variable that we will use to indicate that the window has already been created.

We really haven't done anything yet. We have just told the class that it has a couple of extra things in it. To add the necessary code, close and save the mainfrm.h file and open the mainfrm.cpp file. At some place in this file, add the following code:

```
void CMainFrame::ActivateFrame(int nCmdShow)
{
   CRect rect;
   WINDOWPLACEMENT wndpl;

   if (m_bFirstTime)
   {
      rect.left = 0;
      rect.top = 0;
      rect.right = 500;
      rect.bottom = 480;

      wndpl.length = sizeof(WINDOWPLACEMENT);
      wndpl.showCmd = SW_NORMAL;
      wndpl.flags = 0;
      wndpl.ptMinPosition = CPoint(0,0);
      wndpl.ptMaxPosition = CPoint(-::GetSystemMetrics(SM_CXBORDER),
                            -::GetSystemMetrics(SM_CYBORDER));
      wndpl.rcNormalPosition = rect;

      BOOL bRect = SetWindowPlacement(&wndpl);
   }
   CFrameWnd::ActivateFrame(nCmdShow);
}
```

This code is the function that we are overriding. It will be explained in a moment. There is one additional code change, and this takes place in the CMainFrame::CMainFrame() function. Change the function to look like the following code:

```
CMainFrame::CMainFrame()
```

```
{
        // TODO: add member initialization code here
        m_bFirstTime = TRUE;
}
```

Look back at the ActivateFrame() function that was added to the CMain-Frame class. What is basically happening in this function is that we are changing a global data structure that keeps track of the size and placement of the window. When this code is executed, the window hasn't been created on the screen yet. Since we only need to do this once, we have the *m_bFirstTime* variable. If this variable is TRUE, then this is the first time that we have seen this code.

The code sets the size of the window with the statements

```
    rect.left = 0;
    rect.top = 0;
    rect.right = 500;
    rect.bottom = 480;
```

Notice that the size of the window is only 500 × 480. The reason for this is that a second window will be created in Chapter 6 that sits just to the right of the main window. Adding this second window means the primary window needs to be smaller than 640 so that there is no overlapping.

The remainder of the code in this function sets the placement of the window. The last line in the bracketed code calls the function SetWindowPlacement() to activate the changes we have made to the default window data structure. The last line in the function makes a call to the default ActivateFrame() method of the CMainFrame class. This guarantees that our code is executed and all of the code of the CMainFrame class is executed as well. If we were to leave off this last statement, there is a good chance that the application wouldn't have something necessary for window creation.

To see the changes in our application, Build the project and Execute it. What you should find happens is that a slightly oblong window appears on the screen in the upper left corner. This occurrs because we told the system exactly where to start the window and its size.

The application code created so far can be found in the directory netwar2/chapt5/a on the enclosed CD-ROM.

Adding a Background Bitmap

With the basic application foundation completed, it's time to start adding things so that the application begins looking more like our game. In this section, we will look at adding a background bitmap. This background bitmap will be the map that the kings will move around on to defend their realm. The map is not an essential part of the game. It's true purpose is to give the game a little graphical splash. You can edit the bitmap that we supply or add your own based on the techniques discussed here.

The bitmap we are going to use is called backgrnd.bmp and is located along with the code from this section in the directory netwar2/chapt5/b on the enclosed CD-ROM. You can use any paint program to change the look of the bitmap.

Loading the Bitmap using AppStudio

We have a bitmap on the disk, and we want to use it in our application. In order to keep things consistent in the Visual C++ system, we should use the AppStudio application to add the bitmap into our game as a resource. Execute the AppStudio from the Visual C++ menu by clicking on Tools and AppStudio. When AppStudio displays its window, you will see that AppWizard has already created a couple of resources for the application. We need to Import a resource.

The background bitmap is going to be Imported because the bitmap editor supplied with AppStudio cannot handle 256 color bitmaps. To do the Import, click on Resource and Import. You will be prompted with an OPEN common dialog box. The background bitmap file should automatically appear in the left file box. Double-click on the file name to select it.

AppStudio will display a message saying that it cannot display the bitmap because the editor does not support 256 colors. Click on the OK button to continue. After clicking on OK, a BITMAP type will appear on the left-hand side of the dialog box in the AppStudio window. The right-hand side of the dialog box will have the name of the bitmap resource chosen by AppStudio. We need to change this name so that it is something we can use to identify this bitmap as the background bitmap.

Click on the Properties button at the bottom of the current dialog box. A new dialog box will appear. On the right part of the box will be a small picture of the background bitmap. If this is not the case, delete this resource and try to Import the background bitmap again.

In the ID: section of the dialog box, type the name "IDB_PLAYING_FIELD" and press <Return>. The background bitmap has been successfully added as a resource in our application. Now it's time to load the bitmap into our application.

Loading the Bitmap into the Application

When our application is executed, there will be several things that we want to initialize. This initialization process will need to be done only once. In fact, some of the things that will be initialized can only be done once. This means that we need some kind of mechanism to execute this code once. Thanks to the nature of C++, we have just the mechanism.

Open the netwavw.cpp file and find the entry for CNetwar2View::CNetwar2View(). This is the constructor for the View class in our application. The constructor is only executed at the start of the application when an object of this View class is invoked. This means the constructor will only be executed once for this particular application. This is the perfect place to put some initialization code.

To display the background bitmap, we have to create a compatible device context to hold the bitmap so that it can be copied to our application's window. The bitmap will also have to be read in from the disk. Both of these operations require a variable to hold their respective structures. Therefore, we need to first declare the variables in the header file for our view.

In the file netwavw.h, add the following to the class structure:

```
private:
    CDC*        m_pDisplayMemDC;
    CBitmap*    playing_field_bitmap;     // our playing field
```

Adding this code to the header file and our View class will allow any method associated with the View class to access the variables.

Now that we have the variables added, it's time to use them. In the netwavw.cpp file, add the following code in the CNetwar2View:: CNetwar2View() function:

```
m_pDisplayMemDC = new CDC;
CClientDC dc(this);
OnPrepareDC(&dc);
m_pDisplayMemDC->CreateCompatibleDC(&dc);

// Load Playing field bitmap
playing_field_bitmap = new CBitmap;
playing_field_bitmap->LoadBitmap(IDB_PLAYING_FIELD);
m_pDisplayMemDC->SelectObject(playing_field_bitmap);
```

The code is broken down into two parts: initialization of the application's device contexts and loading the bitmap. We start with the device contexts.

Initializing Device Contexts

The device context we declared in the header file is for the compatible device context. A new DC object needs to be created for the variable. This is performed with the code:

```
m_pDisplayMemDC = new CDC;
```

As we saw in a previous chapter, the MFC class for a device context is called CDC, and using the *new* statement with a class will create a new CDC object and associate it with the device context we declared. Since this device context is supposed to be compatible with a device context for our application, one had better be obtained. The code

```
CClientDC dc(this);
OnPrepareDC(&dc);
```

does the appropriate job. The first line creates a device context associated with the client area of the application's window. Using CClientD will automatically create and release the device context. The programmer does not have to worry about these operations.

The second line of code doesn't do much for standard applications such as ours. If your application has scroll bars and other controls, OnPrepareDC() will adjust the attributes of the device context to take these controls into consideration. Now that the application's device context has been initialized, the compatible device context can be set up. This is done with the code:

```
m_pDisplayMemDC->CreateCompatibleDC(&dc);
```

The CreateCompatibleDC will create a device context associated with the *m_pDisplayMemDC* device context object.

Initializing the Background Bitmap

The device context will allow us to draw to the screen. The background bitmap is the thing that will be drawn to the screen, but it must be read into the system. The background bitmap is a system bitmap. A variable called playing_field_bitmap has been created of type Cbitmap. Cbitmap is an MFC class specifically for bitmaps. Before a bitmap can be read and associated with the object, we have to create an instance. The code

```
playing_field_bitmap = new Cbitmap;
```

does the job. With the object created, a bitmap can be read with the code

```
if (playing_field_bitmap->LoadBitmap(IDB_PLAYING_FIELD) == FALSE)
  MessageBox("Unable to load bitmap", "error", MB_OK);
```

The method LoadBitmap() is part of the Cbitmap class and requires a single parameter. The parameter is the ID that we assigned to our bitmap in AppStudio. Once the bitmap is read in, we can associate the bitmap with our compatible device context. The following code does this for us.

```
m_pDisplayMemDC->SelectObject(playing_field_bitmap);
```

According to the Help files of Visual C++, a bitmap can only be selected into a device context and the device context must be set up to accept the bitmap.

If you did any editing to the background bitmap, you will find that the bitmap is just the right size to fit in the 500 × 480 window our application creates. Therefore, the entire bitmap fits into the compatible device context.

Displaying the Bitmap

When we did drawing on the Windows application in previous chapters, we used the OnPaint message handler. In Visual C++, we use the OnDraw()function. The OnDraw() function is called when the Windows system tells the application that the window needs to be redrawn. As we will see later, we can also tell the window to redraw itself.

Add the following code to the OnDraw method in the netwavw.cpp file.

```
BITMAP bm;
playing_field_bitmap->GetObject(sizeof(bm), &bm);
pDC->BitBlt(0,0,bm.bmWidth, bm.bmHeight, m_pDisplayMemDC, 0, 0, SRCCOPY);
```

The first line declares a variable of type *BITMAP.* This data structure will be used to hold size information about the bitmap we are going to display on the screen. The second assigns the information about the background bitmap to the *BITMAP* variable.

The last line does the actual drawing of the background bitmap to our application's window. The BitBlt() function is a method associated with the device context that is sent to the OnDraw() function automatically. Notice we don't have to Get or Release the device context. The function that calls the OnDraw() function does all of this for us.

The fields of the BitBlt() functions are:

first, second=upper left corner to start drawing

third, fourth=width and length of the bitmap to draw

fifth=the device context to draw the bitmap from (this is our compatible device context that the bitmap was selected into)

sixth, seventh=the upper left corner of the source bitmap

eighth=flags that tell how to draw the bitmap

Testing the Game

The last thing we have to do is test our application to make sure that the changes made are correct. All of the new code is located in the netwar2/chapt5/b directory. When you execute the application, you will get the window shown in Figure 5.3. Things are moving along nicely. Let's add more graphics to the screen by designing our playing pieces sprites.

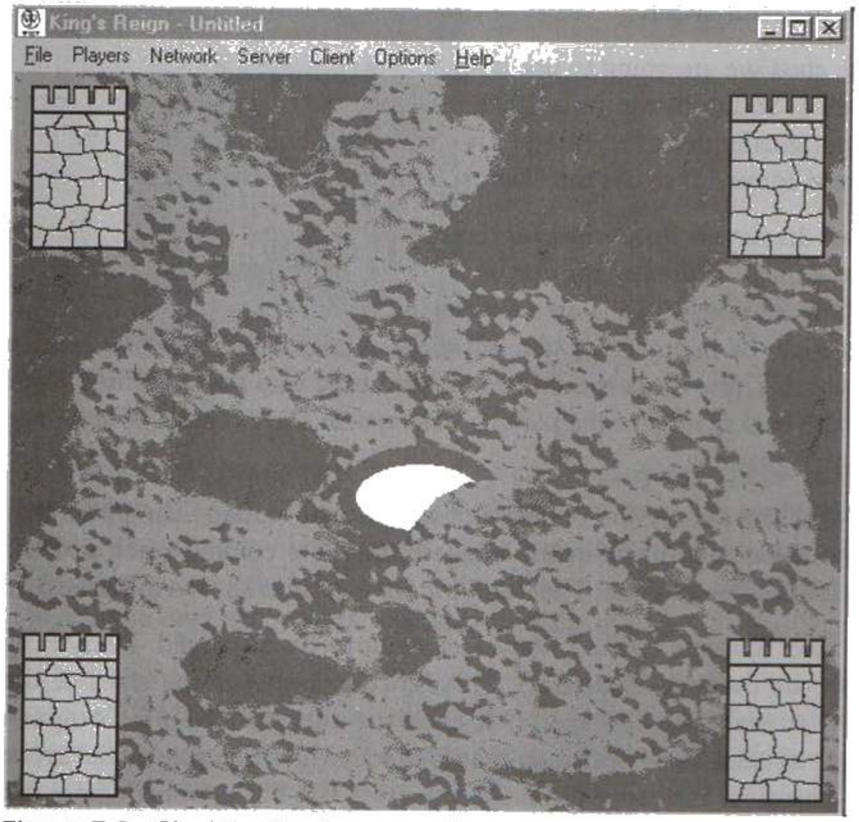


Figure 5.3 *Checking the changes made.*

Sprites

Each of the kings in our game will be able to purchase and move playing pieces. The playing pieces represent the different armies available. Each of the playing pieces has a different look and characteristics. The basic format and proposed use for the playing pieces suggests they make perfect candidates for objects. In this section, a complete sprite library will be discussed as well as the incorporation of this library into a CSprite class that will handle all of the details of our playing pieces.

How the Sprites Work

The Sprite class we are going to use is an adaptation of a Sprite class detailed by Michael J. Young in the June 1993 issue of *BYTE Magazine.* This sprite class was originally designed to add C++ sprites to your programs. We take the class and convert it to use the Microsoft Foundation Class library.

Sprites are displayed in a window using two separate bitmaps called an image and a mask. The bitmaps can be any size, but typically they are small, such as 32 × 32 pixels. In the area for the image bitmap, a picture is drawn. The picture can be in any color you wish. There may be parts of the bitmap that are white. These areas should be colored black. The pixels colored black will have the background of the playing field visible. An example of an image is shown in Figure 5.4.

The mask will be exactly the opposite of the normal bitmap. The pixels that are colored other than black in the image bitmap are colored black and the black pixels are colored white. The mask for our bitmap is shown in Figure 5.5.

With the two bitmaps created, they are ready to be displayed. Everything starts with the function Initialize(). This function retrieves the sizes of the bitmaps that make up the sprite. It calls the function CreateCompatibleBitmap() to create a new bitmap that is exactly the same size as the sprite bitmaps. This new bitmap will be used to store the background where the sprite will be displayed.

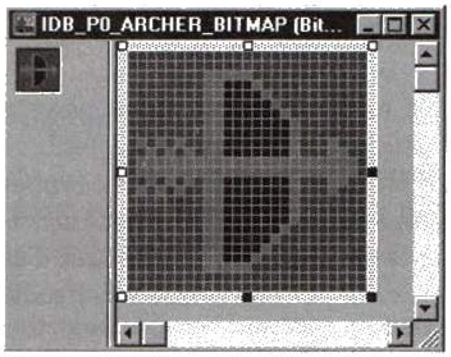


Figure 5.4 *Example of the pixels.*

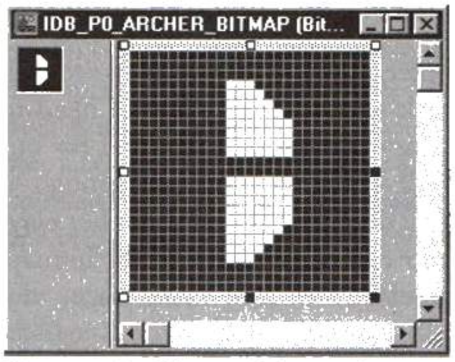


Figure 5.5 *The mask for our bitmap.*

Once the sprite has been initialized, it can be displayed. The function that first displays the sprite is Start(). This function starts by copying the background to the sprite. The copy is performed using a flag called SRCCOPY. This flag indicates that the copy should take each pixel from the source and put it in the destination with no changes to the pixels.

Now the function could copy the image bitmap to the window, but the function used for the copy BitBlt() only copies rectangular blocks of pixels. This is where the mask and image bitmaps come into play. The sprite is copied to the window using two copies.

The first copy copies the mask bitmap to the window using a flag called SRCAND and the BitBlt() function. This will cause the white pixels of the mask to display the background pixels and the black pixels to be displayed on the window as is. This means only the black part of the mask will be displayed on the window. The surrounding pixels of the sprite will be displayed as the background.

Now a copy is made with the normal bitmap and a flag called SRCINVERT. This copy causes the pixels to be copied using the XOR operator. All of the black pixels on the image bitmap will be displayed as the background, the colored pixels will be XORed with the black pixels, and the colored pixels will win out.

Doing this sequence each time the sprite is moved and replaced the background that was previously saved gives the effect of moving the sprite over a background image.

Drawing the Pieces

As we have just read, a single playing piece represented as a sprite will require a minimum of two bitmaps: mask and original. We will need to design these bitmaps ourselves. There is another bitmap required for our game however. If you go back to Chapter 2 where we did a run-through of the game, you will notice that when a player double-clicks on his or her playing piece, it becomes highlighted with yellow in certain parts of the bitmap. This is a third bitmap that will have to be created for each playing piece.

Another topic that needs to be discussed concerning our playing pieces and the sprite bitmaps is how big or small to make them. In the game supplied on the enclosed CD-ROM, the bitmaps for the playing pieces are 25 pixels by 25 pixels. This size was chosen based on two factors. The first was the size of the playing piece on a 640 × 480 screen. I created the game on a 1024 × 768 screen, which causes the playing field and pieces to be much smaller than the 640 × 480 screen. The original size of the bitmaps was 50 × 50. On a 1024 × 768 screen, the playing pieces looked just right, but when I resized my screen to 640 × 480, the playing pieces were huge. In fact, each playing piece took up a good part of the background. This meant the size of the bitmaps had to be reduced. The second factor was the game played by 1024 × 768. The pieces couldn't be so small that they couldn't be controlled with the mouse cursor. All of this led to the 25 × 25 size.

Bitmap Drawing

The actual creation of a playing piece can be an artistic challenge in itself. There are several different techniques that you can use to create the bitmaps. The first is to take an image, shrink it and Import it into AppStudio just as we did for the game's background image. The problem with this is that the detail in the original image is usually lost and the time spent bringing some of the detail back in the 25 × 25 bitmap could be used to make a new bitmap image from scratch.

The second technique is to find a friend or hire someone who has artistic talent and can use a computer paint program. The paint program to use is in the AppStudio application itself. Although primitive, it includes all of the necessary tools, especially when dealing with a 25 × 25 bitmap.

Let's look at creating a bitmap for the cavalry division playing piece. Start AppStudio from the Tools menu item. Click on the Bitmap type in the dialog box that appears in AppStudio. Now click on the NEW button at the bottom of the box. A small dialog box will appear asking you which type of resource to create. Click on OK, since Bitmap should be highlighted. If it is not highlighted, double-click on Bitmap. You should see a new dialog box with a gridded area in it and a dialog box with tools and colors in it (Figure 5.6).

The gridded area is where the bitmap will be drawn. The first thing to do is change the size of the bitmap. Click on the Resource menu item and then Properties. In the edit lines for X and Y size, change the values to 25 and double-click the dialog's Close box. The gridded area will shrink considerably. This is the area that you have to work with if you want your playing pieces to be 25 × 25. (This isn't going to be a tutorial on using the bitmap editor in AppStudio, so refer to the online help for help in using the tools and colors in the tool dialog box.)

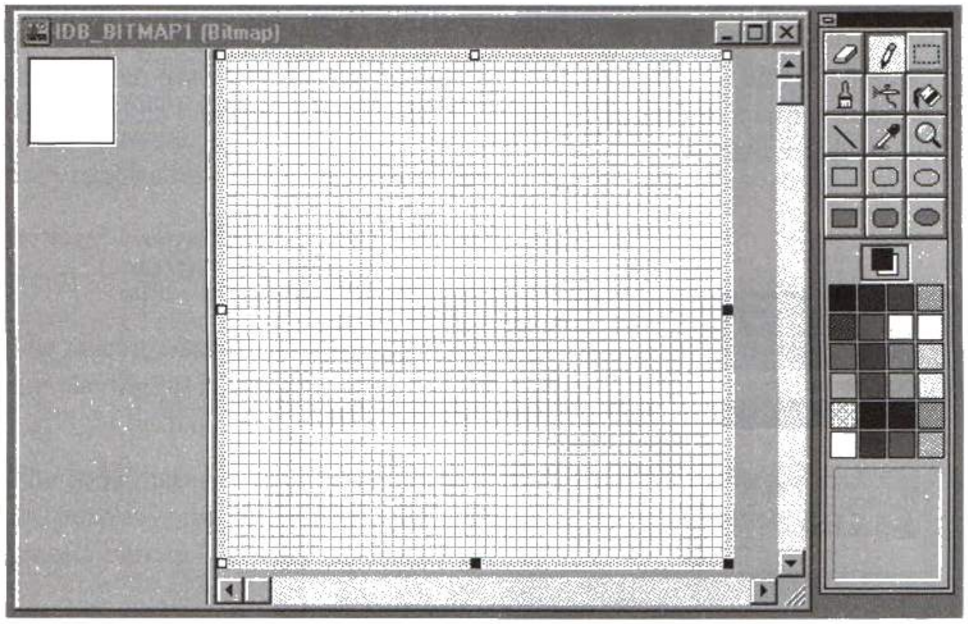


Figure 5.6 *Creating a bitmap for the cavalry division.*

Once your bitmap has been drawn, double-click on the Close button on the window with the bitmap in it. After the window closes, a new entry will appear in the Resource Script window of AppStudio. The name of the bitmap will be something like IDB_BITMAP1. Click on Properties and change the name to IDB_CAVALRY_BITMAP.

That's all it takes to create the playing pieces in our game except for one thing. We didn't create the mask for the bitmap. The mask for each playing piece has its pixels black for every pixel in the image that is not white. We also have to go back and change the playing piece image so that all white images are black.

The easiest way to do this is to go back and change the playing piece image so all of the white pixels are black and close the bitmap. Do this by double-clicking on the name of the bitmap you want to edit. The two dialog boxes used to create the bitmap will appear on the screen. Do the necessary editing and close the bitmap.

Now click on the name of our cavalry bitmap. Click on the Edit menu item and Copy. Now click on Edit again and then Paste. A new entry will appear that is something like IDB_CAVALRY_BITMAP2. Click on Properties and change the name to IDB_CAVALRY_BITMAP_MASK. Now we need to edit this mask bitmap so that all of the black pixels are white and all of the colored pixels are black.

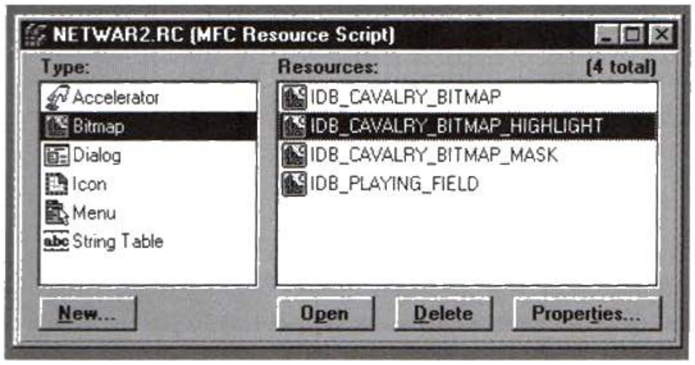


Figure 5.7 *The four bitmaps.*

Once these steps are completed, we have a complete playing set. Well, almost. We need a third bitmap, remember? This bitmap is for the highlighted playing piece. The good news is we can use the same mask as long as we add the yellow highlights to the pixels that have been previously colored. To create this bitmap, click on the IDB_CAVALRY_BITMAP bitmap and then copy and paste a new one. Change the name of the new bitmap to IDB_CAVALRY_BITMAP_HL. Edit the bitmap and place several areas of yellow on the pixels that have already been colored. Close the bitmap when finished. When you are done, you should have four bitmaps in your resource file. You can now save and close the AppStudio application.

Adding the Pieces to the Game

As in the case of the background bitmap, we have to add the pieces to the game. Although the bitmaps are in the resource file and will be compiled in our .exe file, they have to be loaded into the system in order to be used. Since the bitmaps will be loaded only once, the most logical place for the load is in the constructor for the application's view object.

The code for loading the bitmaps is exactly the same as used for the background bitmap except we will need a few more variables to hold the handles to the bitmaps. In the netwavw.h file, add the following variables right under the variables for the background bitmap.

```
CBITMAP*  cavalry_bitmap,
          cavalry_bitmap_mask,
          cavalry_bitmap_highlight;
```

The *cavalry_bitmap* variable will be an instance of a BITMAP and will relate to the playing piece's bitmap image. The other two variables are the mask and highlight bitmap.

The code that uses these variables is added to the view object constructor in the netwavw.cpp file. Add the following after the code for loading the background bitmap.

```
//Load bitmaps
   cavalry_bitmap = new CBitmap;
   if (cavalry_bitmap->LoadBitmap(IDB_CAVALRY_BITMAP) == FALSE)
```

```
      MessageBox("Unable to load bitmap", "error", MB_OK);

   cavalry_bitmap_mask = new CBitmap;
   if (cavalry_bitmap_mask->LoadBitmap(IDB_CAVALRY_BITMAP_MASK) == FALSE)
      MessageBox("Unable to load bitmap", "error", MB_OK);

   cavalry_bitmap_highlight = new CBitmap;
   if (cavalry_bitmap_highlight->LoadBitmap(IDB_CAVALRY_BITMAP_HL) == FALSE)
      MessageBox("Unable to load bitmap", "error", MB_OK);
```

This the exactly the same code used for loading the background bitmap. In each case, an object is instantiated and assigned to the appropriate variable. The method LoadBitmap() is used to load the bitmap from the disk (or in our case the .exe file). The bitmap is loaded based on the AppStudio ID we assigned to each bitmap. If the result of the load is FALSE, the bitmap was not loaded correctly and the user will be given a message indicating the situation.

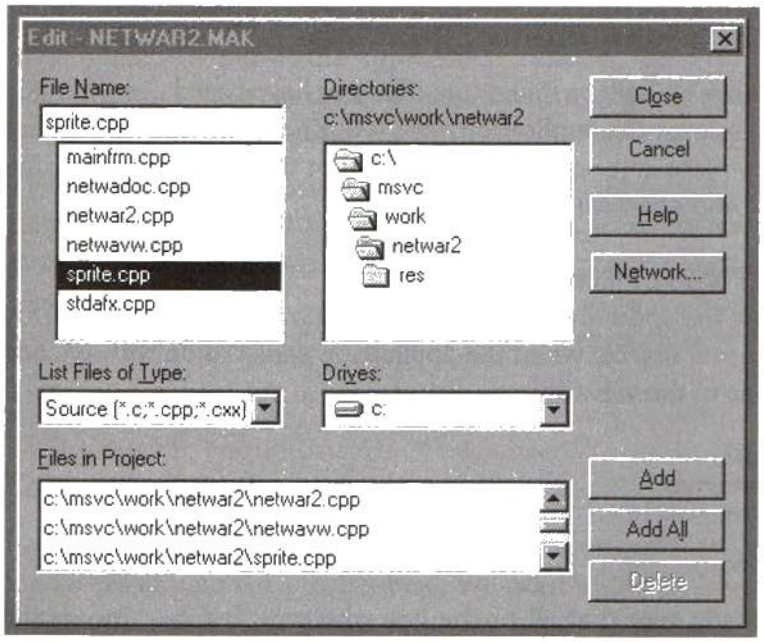
After the third bitmap is loaded, everything is set for using the bitmaps in a playing piece. Now we have to examine the routines that will create and manipulate the bitmaps.

Sprite Library Routines

In an earlier section, we went over the details of a sprite library that was modified for use in the game. This library includes the necessary routines for creating and manipulating bitmaps into sprites. The sprites will be used to represent the playing pieces in our game. The routines that are available are:

```
CSprite ();
~CSprite ();

void GetCoord (CRect *Rect);
BOOL Hide (CDC* HDc);
BOOL Initialize (CDC* Hdc,
                 CBitmap* HMask,
                 CBitmap* HImage,
                 CBitmap* HHighlight);
BOOL MoveTo (CDC* HDc, int X, int Y);
BOOL Redraw (CDC* HDc);
BOOL Start (CDC* HDc, int X, int Y);
BOOL ReplaceBitmap();
```

In order to use the routines, we have to add the sprite library to our project. To do this, click on Project and Edit. You will be presented with the dialog box shown in Figure 5.8.



```
c:\msvc\work\netwar2\netwar2.cpp
c:\msvc\work\netwar2\netwavw.cpp
c:\msvc\work\netwar2\sprite.cpp
```

Figure 5.8 *Add the sprite library to our project.*

Find the filename sprite.cpp in the upper-left listing of files. Double-click on this name to add it to the project file. Click on Close. The system will do some housekeeping and return you to the Visual C++ WorkBench.

Adding a sprite to our game is a two-step process. First, the sprite is initialized, and second, it is displayed on the screen. The first step will be performed once, so it can be done in the view object's constructor. We will need a variable to keep track of this sprite so add the following code to the netwavw.h file at the very top:

```
#include "sprite.h"
```

Add this code in the View class definition after the variables for the bitmaps:

```
CSprite* cavalry_piece;
```

The following code should be added after the bitmaps have been loaded in the constructor:

```
cavalry_piece = new CSprite;
if (cavalry_piece->initialize(GetDC(), cavalry_bitmap_mask, cavalry_bitmap,
cavalry_bitmap_highlight) == FALSE)
   MessageBox("Unable to initialize sprite", "Error", MB_OK);
```

The project can now be built to make sure that you have typed everything in correctly. If you execute the application, you will find that nothing happens as far as a playing piece is concerned. The reason for this is we have only initialized the sprite and not displayed it on the screen.

Destroying the Playing Piece and Sprites

Before we display the sprite in the window, we shouldn't forget to destroy the bitmap and sprite objects when the application ends. To do this, we add the following code to the view object's distracter:

```
delete cavalry_bitmap;
delete cavalry_bitmap_mask;
delete cavalry_bitmap_highlight;

delete cavalry_piece;
```

This code will made sure that all of the housekeeping for the bitmap and sprite objects concerning destruction is taken care of.

Displaying the Playing Piece

We have a playing piece initialized. Now we need to get the piece displayed on the screen. The process is fairly simple. Add the following code to the OnDraw() function in the netwavw.cpp file:

```
if (cavalry_piece->show)
    cavalry_piece->Redraw(pDC);
 else
 {
    cavalry_piece->Start(pDC, 100, 100);
    cavalry_piece->show = TRUE;
 }
```

This is all of the code necessary to draw our playing piece on the screen. It starts out by examining the *show* PUBLIC variable of our playing piece Csprite object. If this field is FALSE, then this sprite has never been drawn on the screen before now. If the field is TRUE, then the system knows that playing piece is already on the screen and it only needs to redraw it.

The distinction is important. In the FALSE case, a call is made to the Start() method of the object. This method requires the device context to use for displaying the sprite as well as the X and Y position to place the piece. After the playing piece is put at the appropriate location, the *show* variable is set to TRUE so that the next time the OnDraw() method is called, the sprite will not be placed at the starting location again, but will only be redrawn.

The sprite is only drawn when *show* is TRUE. In this case, the method Redraw() is called and the sprite is redrawn at its current location. At the moment this distinction might not seem important, but in a later section, Mouse Movement and Clicking, we are going to be moving the sprite around and we always want the sprite to be redrawn at its present position.

The code for the game up to this point is in the directory /netwar2/chapt5/c.

Complete Sprites

Before we look at adding mouse support to our game and moving our playing pieces around the playing field, we need to discuss two topics. The first deals with the playing pieces necessary for our game. We are designing this game for four players. Each of the players will see the same playing field and pieces on their screen. There isn't any problem with this and the background bitmap, but what about the playing pieces? Will all of the players know which playing pieces are theirs? If all of the playing pieces look the same, probably not. However, we don't want to change the pieces too much, since players should know what kind of opponent they are attacking.

One solution is to color code each of the player's pieces. This is the technique used in the game on the CD-ROM. Each of the players is a different color. The playing pieces are drawn such that the background of the playing piece is the color of the player. This means you can tell at a glance where your pieces are and where the pieces of your opponent are.

Therefore, each player will have four playing pieces for each of the different armies: cavalry, flyer, legion, and archer. That's a total of 16 playing pieces. All of the playing pieces will be the same, so only one mask is necessary: a bitmap of all black pixels. We also have the highlighted playing pieces to consider as well. This will be 16 more playing pieces. This means we will have a total of 33 playing pieces and bitmaps to draw and create. All of the bitmaps and the variables to control them will be added in a later chapter. I just wanted you to be aware of the number of bitmaps that will need to be created for the game.

Advanced Sprites

The other topic I wanted to touch on here was animated sprites. You could create more than one image for each bitmap. Each time the OnDraw() method is executed, the different bitmaps could be rotated and an animated bitmap will appear on the playing field. I will leave this as an exercise for the reader, as it is beyond the scope of this book.

Making Sprites into Playing Pieces

The sprites work fine as playing pieces except that there are a couple of additional variables that each playing piece will need. Add the following code to the Csprite class definition in the sprite.h file:

```
int defense,
    offense,
    player,
    speed;

char  type[10];

BOOL  active,
        war;
```

All of these variables need to be initialized, so add the following code to the Csprite class constructor in the sprite.cpp file:

```
defense = 0;
offense = 0;
player = 0;

active = FALSE;
war = FALSE;

strcpy(type, "");
```

As we mentioned earlier, each playing piece will have a strength to it. This strength will be measured in both offense and defense points. The points will be used when a war is initiated by any of the users.

There are a total of four different playing pieces in the game. The *type* variable is used to hold the name of the playing piece. Since several players will be using the same playing field at the same time, we have to keep track of which player owns which playing pieces. The variable *player* will be used for this purpose. It will also serve as a check when a player clicks or double-clicks on a playing piece. Players should not be allowed to move playing pieces that are not theirs.

The playing pieces are designed to have varying degrees of defense and offense capabilities. At the same time, the speed of the pieces varies. The variable *speed* is used to determine how far each piece can move at a time.

In Chapter 6, we will see that our playing piece will be created, but not immediately placed on the playing field. For this reason, we will need to have a variable called *active* that tells us whether or not the current playing piece has been activated and placed on the playing field.

The last variable added is called *war*. This variable is set to TRUE when the current playing piece has been selected for war by any of the players. This is an important variable, since we don't want playing pieces to be moved that are currently selected for war.

With all of these new variables, we need to make a change to our sprite library. When the sprite is initialized, we had better specify all of these new variables so the sprite is created correctly. In the file sprite.h, change the header for Initialize from

```
BOOL Initialize (CDC* Hdc,
                        CBitmap* HMask,
                        CBitmap* HImage,
                        CBitmap* HHighlight
                        );
```

to

```
BOOL Initialize (CDC* Hdc,
                        CBitmap* HMask,
                        CBitmap* HImage,
                        CBitmap* Hhighlight,
                        BOOL Active,
                        int Offense,
                        int Defense,
                        int Player,
                        char *Type,
                        int Speed
                        );
```

In the sprite.cpp file, add the following code to the end of the sprite constructor:

```
defense = 0;
offense = 0;
player = -1;
speed = 0;

active = FALSE;
```

Add the following code to the Initialize() function in the sprite.cpp file:

```
active = Active;
offense = Offense;
defense = Defense;
player = Player;
strcpy(type, Type);
speed = Speed;
```

Also remember to change the function header for the Initialize() function. The last thing you need to do is open the netwavw.cpp file and change the call to the Initialize() function to include all of the new information about a sprite. You can make up the information for the moment.

Mouse Movement and Clicking

At this point, we have a background on our window and a single playing piece ready to do something. By far the easiest way to control the playing piece would be to use the mouse. This means our game must be able to respond to mouse commands such as movement and clicks. Visual C++ makes responding to the mouse a breeze.

The mouse and its associated messages are part of the View class. We will use the ClassWizard to set them up for us. At the Visual C++ WorkBench, click on Browse and ClassWizard. Make sure that the class in the Class Name field is CNetwar2View. In the Object ID's list box will be an entry for CNetwar2View. Click on this entry. A list of messages will appear in the right list box.

Go through the list of messages and add WM_LBUTTONDOWN, WM_LBUTTONDBLCLK, WM_LBUTTONUP, and WM_MOUSEMOVE.

Once all three of these messages are listed in the bottom list box, click on the OK button. Now open the netwavw.cpp file. At the bottom of the file will be the basic structures for these messages. Remember from Chapter 4 that Visual C++ automatically sets up the messages and their handlers so that all we have to do is add code to the handler routines. All of the housekeeping for messages is taken care of for us.

Left Mouse Button Click

The first message we are going to deal with is the left mouse button click. When the left mouse button is clicked, the flow of control in the program will be sent to the function called CNetwar2View::OnLButtonDown(UINT nFlags, CPoint point). The system sends a number of flags to the function as well as the current location of the mouse represented in the variable *point*. *Point* is a C++ object from the Cpoint class. It has two fields associated with it: X and Y.

When the left mouse button is clicked, this means that the user is clicking a control of one of the application's windows or is trying to select a playing piece to move. In the case of clicking a control such as a menu item, your application does not activate the OnLButtonDown() function. This function is only called when the user clicks the mouse somewhere in the client region of the view object's window. Therefore, we know that when this function is called, the user is trying to move a playing piece.

We only want to allow the user to move a playing piece when the mouse cursor is actually touching the playing piece itself. The user shouldn't be able to move playing pieces when the cursor is nowhere near the piece.

In the netwavw.cpp file, find the OnLButtonDown() method and make it look like the code that follows.

```
void CNetwar2View::OnLButtonDown(UINT nFlags, CPoint point)
{
 CRect* SpriteRect;
 CRect clientRect;
 CPoint topleft;

 SpriteRect = new CRect(cavalry_piece->mX, cavalry_piece->mY,
    cavalry_piece->mX+cavalry_piece->mWidth, cavalry_piece->mY+cavalry_piece-
>mHeight);
```

```
  if (SpriteRect->PtInRect(point))
  {
    SetCapture();

    GetClientRect(clientRect);
    MapWindowPoints(NULL, clientRect);
    topleft = SpriteRect->TopLeft();

    cavalry_piece->XOffset = point.x - topleft.x;
    cavalry_piece->YOffset = point.y - topleft.y;

    dragging = 1;
  }
  return;
}
```

This is all of the code necessary to first detect when the user has clicked on a playing piece and to set things up for moving the playing piece when the user has clicked on it. The function starts with three local variables. The first, called *SpriteRect,* is an object pointer. It will be initialized shortly. It is used to hold a rectangle the size of the sprite. The second parameter is a Crect object called *clientRect.* It will be used to hold a rectangle the size of the client region of our view object's window. The third parameter is a Cpoint object called *topleft.* It will be used to hold the topleft corner of our sprite playing piece.

After the declaration of the variables, the *SpriteRect* object is instantiated and initialized to the dimensions of the sprite playing piece. The Crect class has a method called PtInRect() that takes as a single parameter an X, Y coordinate pair in the form of a Cpoint object. The next statement of code uses this function along with the dimensions of the playing piece and the current location of the mouse when the left mouse button was pressed. If the PtInRect() method returns TRUE, then we know that the user has clicked on the playing piece. If the method returns FALSE, the user clicked somewhere else in the client region. We are most concerned when the user clicks on a playing piece.

The first line of code in the TRUE part of the IF statement is *SetCapture().* This function call tells the Windows system that our application would like a message each time the mouse is moved. The message is called WM_MOUSEMOVE. The next couple of lines sets the client region of the window for mouse movement translation and determines the offset from the top-left corner of the sprite to the exact location in the sprite where the user pressed the left mouse button. This ensures that the playing piece is moved correctly and does gain or lose position on the screen.

The last thing that happens when a user clicks on a playing piece is that a global variable called *dragging* is set to 1. This variable will be used to indicate to other message handlers that a playing piece is currently being dragged. You will need to add this variable to the netwavw.h file. Add the variable as

```
int dragging;
```

just under the declaration of the *cavalry_piece* variable in the view class definition. Since we will be using this variable to indicate that a playing piece is being dragged, we had better initialize it when the application starts. In the constructor for our view object, add the code:

```
dragging = -1;
```

The value of −1 will be used to indicate a playing piece is not being dragged.

Mouse Move

The user has clicked on a playing piece and is starting to move the mouse. Since we told the Windows system to let us know about mouse movements, we had better respond to them. The message handler void CNetwar2View::OnMouseMove(UINT nFlags, Cpoint point) is where the response takes place. Locate this handler in the netwavw.cpp file and make it look like the following code:

```
void CNetwar2View::OnMouseMove(UINT nFlags, CPoint point)
{
  int move_x, move_y;
  if (dragging == -1)
   return;

  move_x = (point.x - cavalry_piece->XOffset);
  move_y = (point.y - cavalry_piece->YOffset);

  cavalry_piece->MoveTo(GetDC(), move_x, move_y);
}
```

Each time this message handler is called, we know a playing piece is being moved. However, for safety's sake, we had better be sure. The IF statement does a quick check of the *dragging* variable to make sure that it isn't set to −1. Next, two local variables are set to specific X and Y values representing the amount of movement that has taken place by the mouse. The old top-left corner of the sprite is subtracted from the current location of the mouse to obtain the values.

The values are sent as parameters to the Csprite method MoveTo(). This method will move the sprite to the new location and take care of all of the background images the sprite will travel over.

Left Mouse Button Up

The last message handler we have to consider for playing piece movement is when the user releases the left mouse button. This indicates that the user no longer wants to drag this playing piece. The message handler is called void CNetwar2View::OnLButtonUp(UINT nFlags, Cpoint point). Find this function in the netwavw.cpp file and make it look like the following code:

```
void CNetwar2View::OnLButtonUp(UINT nFlags, CPoint point)
{
   if (dragging == -1)
   return;
 ReleaseCapture();

 dragging = -1;
}
```

This message handler is very simple. First a check is made to verify that we are dragging a playing piece. Next, a call is made to the function ReleaseCapture();. This function tells the Windows system that we are no longer interested in messages about mouse movement. The last statement sets the *dragging* variable to −1, thus indicating that no playing piece is currently being dragged.

That's it for mouse messages and dragging our playing pieces. The code up to this point can be found in the /netwar2/chapt5/d directory on the enclosed CD-ROM.

There are two other mouse clicks that we are going to respond to in the game. When users left double-click on a playing piece, they will be selecting the playing piece for war. The playing piece should be highlighted as such. However, if the user double-clicks again on the same playing piece, the piece will be unselected for war and the highlight has to be taken off. The other mouse command will be a right double-click. When the user right double-clicks on a playing piece, a statistics window should appear telling the user about the playing piece. Let's design both handlers now.

Left Mouse Button Double-click

A double-click of the left mouse button causes the WM_LBUTTONDBLCLK message to be sent to our application. Our application will use the handler void CNetwar2View::OnLButtonDblClk(UINT nFlags, Cpoint point). Find this method in the netwavw.h file and make it look like the following code:

```
void CNetwar2View::OnLButtonDblClk(UINT nFlags, CPoint point)
{

   CRect* SpriteRect;

   SpriteRect = new CRect(cavalry_piece->mX, cavalry_piece->mY,
     cavalry_piece->mX+cavalry_piece->mWidth, cavalry_piece->mY+cavalry_piece-
>mHeight);

   if (SpriteRect->PtInRect(point))
   {
     if (current_select == 1)
     {
        current_select = 0;

        cavalry_piece->ReplaceBitmap();
        cavalry_piece->war = FALSE;

        Invalidate(TRUE);
        return;
     }

     if (current_select == 0)
     {
        current_select = 1;

        cavalry_piece->ReplaceBitmap();
        cavalry_piece->war = TRUE;
```

```
            Invalidate(TRUE);
            return;
        }
      }
    }
```

The very beginning of this message handler looks very much like the start of the left mouse down message handler. The code gets the current location of the sprite and checks to see if the user has double-clicked in the area of the playing piece. If the user has, there is more code to execute.

In our game, users must double-click on themselves first and then an opponent in order to do a battle. A global variable will be used called *current_select* that will indicate whether or not players have clicked on themselves as a start to a battle.

If this variable is 0, the user has not clicked on any of his or her playing pieces before. The first thing that we have to do is set the *current_select* variable to 1, indicating that the user has double-clicked on one of his or her playing pieces. To indicate that this playing piece may be used in a war, we highlight the playing piece. We do this by changing the current image of the playing piece. As noted before, the method ReplaceBitmap() does the job for us. This method acts as a toggle. Each time the function is called, the highlighted and normal images are swapped. We make a call to the function to put the highlighted bitmap in place. We also set the sprite variable *war* to TRUE, indicating that this playing piece is getting ready for war.

After the bitmap is switched, the function Invalidate() is called. This tells the system that the view's client region needs to be repainted. The function calls results in a call to OnDraw().

If this variable is 1, the user has clicked on his or her playing piece at some other time. In this case, the user wishes to deselect him- or herself. If a war is not taking place (this will be covered in Chapter 7), users should be allowed to remove themselves from the start of a war. We do the exact same steps as for the variable *current_select* being equal to 0, except that we set this variable back to 0.

Playing Piece Move Update

Now that we are able to select and deselect a playing piece for war, we have to make an update to our movement handlers for the playing pieces. In the handler OnLButtonDown(), change the line of code

```
if (SpriteRect->PtInRect(point))
```

to

```
if ((cavalry_piece->war == FALSE) && (SpriteRect->PtInRect(point)))
```

What we have done is added code that will not allow us to select a playing piece for movement when it is currently selected for participation in a war.

Right Mouse Button Double-click

The right mouse button is going to have a special purpose in our game. If the user double-clicks the right mouse button with the cursor on a playing piece, a statistics dialog box will appear. The dialog box details information about the playing piece, including the type, owner, and offense/defense points. Before we get to the code necessary to activate the dialog box, we'll design the box first, using AppStudio.

With our current project loaded into Visual WorkBench, click on Tools and AppStudio. As usual, AppStudio will display a graphical representation of our resource file. Click on the Dialog type and then the NEW button. Click on OK when the Selection dialog box appears. You should now have the screen on your desktop that is shown in Figure 5.9.

Thanks to the tools provided in AppStudio, creating dialog boxes is very easy. Follow these steps to create our dialog box. We will be working on the dialog box that has the grid points in it.

Step 1: Click on the Cancel button and press the DEL key. The button should be removed.

Step 2: Click on the OK button and drag it to the bottom of the dialog box. Leave a single point between the button and the dialog box bot-

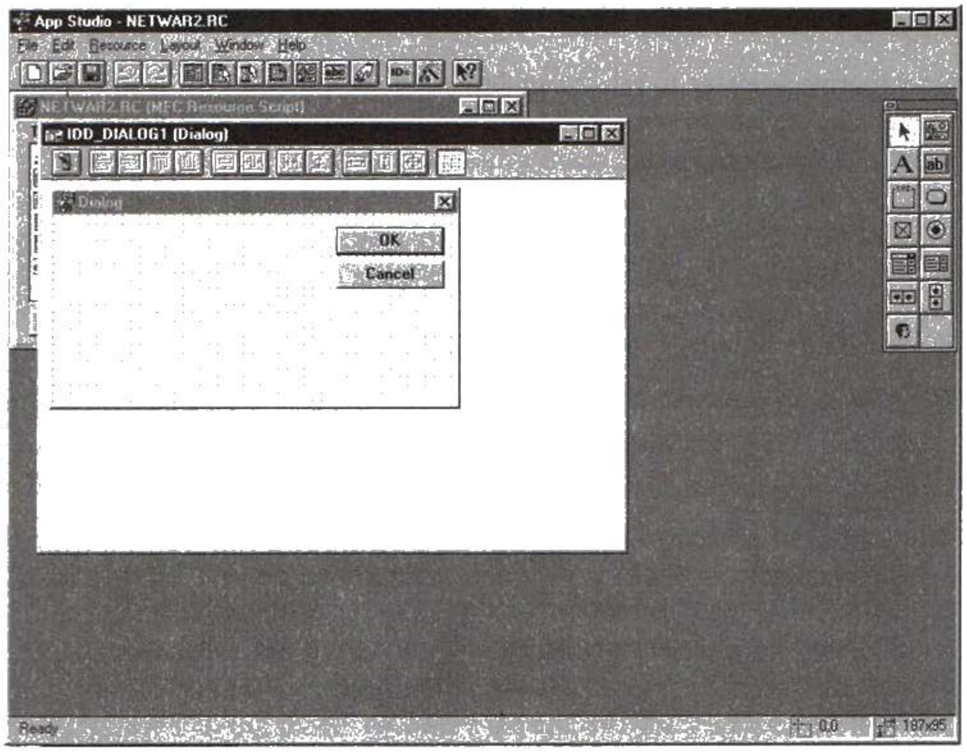


Figure 5.9 *The beginning screen.*

tom. At the toolbar closest to the dialog box we are building, you will see a button with a vertical bar and arrow on either side of the bar. Click this button. Your OK button will be automatically centered horizontally.

Your dialog box should look like Figure 5.10

Step 3: Double-click on the Title bar of our new dialog box. A dialog box will appear with information about our box. In the edit field labeled Caption: type the name "Playing Piece Statistics".

Step 4: In the same information dialog box, find the edit field labeled ID and type the ID string "IDD_PLAYING_PIECE_STATS". Close with information dialog box.

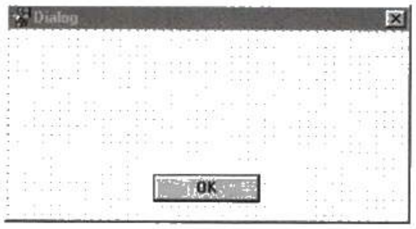


Figure 5.10 *Beginning to design a dialog box.*

Step 5: Click on the button with the large A on it located in the Tools dialog box. This is the text tool. Move the tool to the upper left corner of our new dialog box. Click and drag a box about 40% of the length of the dialog box and tall enough for a single line of text. Release the mouse button.

Step 6: Double-click on the text you just put in the dialog box. Another information dialog box will appear. This dialog box is giving you information about the text control just added. In the ID: edit field type "IDC_STATS_TYPE". In the Caption: edit field type "Type:".

The dialog box should appear as in Figure 5.11.

Notice that the static text we just added has some blank area to the right of it. This will be where the type of playing piece will be put when the user right double-clicks on a playing piece.

Step 7: Add a static text to the right of the TYPE: text. The new text has an ID called "IDC_STATS_OWNER" and a caption of "Owner:"

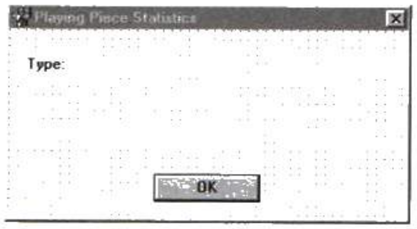


Figure 5.11 *Controlling the text.*

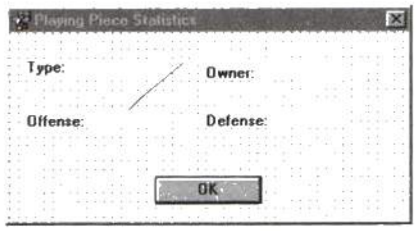


Figure 5.12 *The final dialog box.*

> **Step 8:** Add a static text below the TYPE: text. The new text has an ID called "IDC_STATS_OFFENSE" and a caption of "Offense:"
>
> **Step 9:** Add a static text below the OWNER: text. The new text has an ID called "IDC_STATS_DEFENSE" and a caption of "Defense:"

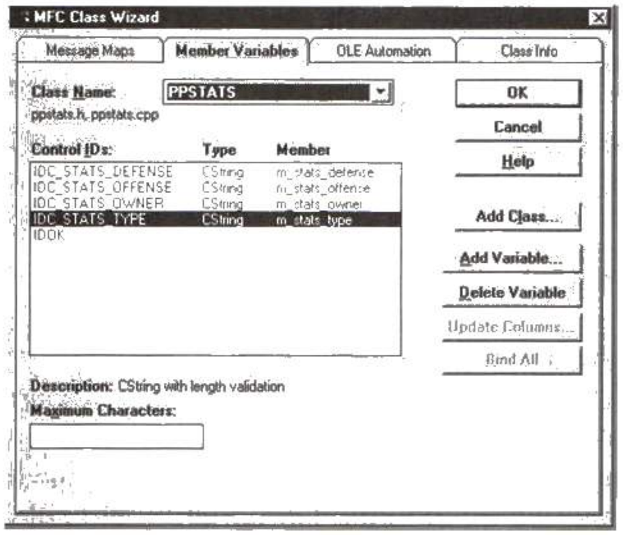
The final dialog box should appear as shown in Figure 5.12.

Using ClassWizard

Creating a dialog box with AppStudio is only a third of the steps necessary for a complete dialog box. The second third of the steps is to use ClassWizard to relate any dialog box controls to our view object. We will start ClassWizard from AppStudio so that it does much of the work for us. Click on the Resources menu item of AppStudio and then ClassWizard.

ClassWizard will start and immediately ask you to name a new class for your dialog box. At the Class Name: edit field, type "PPSTATS" and click on Create Class. ClassWizard will automatically create a new class for the dialog box and put the class definition in the file ppstats.h and the controlling code in the file ppstats.cpp.

After ClassWizard does its housekeeping, it will bring up the Member Variables option and allow you to relate the four text controls to variable in your view object. Double-click on the first entry labeled "IDC_STATS_DEFENSE". An Add Member Variable dialog box will appear. In the first edit field, the dialog box is asking for the variable name to be used for this static text control. Type in "stats_defense" and click on OK. Do the same thing for the remaining three static text controls we added to the dialog box. Don't do



Class Name: PPSTATS
ppstats.h, ppstats.cpp

Control IDs:	Type	Member
IDC_STATS_DEFENSE	CString	m_stats_defense
IDC_STATS_OFFENSE	CString	m_stats_offense
IDC_STATS_OWNER	CString	m_stats_owner
IDC_STATS_TYPE	CString	m_stats_type
IDOK		

Description: CString with length validation
Maximum Characters:

Figure 5.13 *Finishing with ClassWizard.*

anything with the IDOK entry. When you are finished, you should have the screen shown in Figure 5.13.

What we have done is assign specific variables to each of the controls in our new dialog box. Instead of passing messages to and from the dialog box, we just use the variables to assign and change information on the dialog box. Now it's time to use these variables. Click on the OK button of the ClassWizard and save and close AppStudio.

Open the file netwavw.cpp and find the entry for double-clicking the right mouse button. Add the following code:

```
void CNetwar2View::OnRButtonDblClk(UINT nFlags, CPoint point)
{
   PPSTATS      Dlg;
   CRect*       SpriteRect;
   char            string[25];

   SpriteRect = new CRect(cavalry_piece->mX, cavalry_piece->mY,
        cavalry_piece->mX+cavalry piece->mWidth, cavalry_piece->mY+cavalry_piece-
>mHeight);

   if (SpriteRect->PtInRect(point))
   {
        // fill in player
        sprintf(string, "Owner: Player %d", 0);
        Dlg.m_stats_owner = string;

        // fill in type
        sprintf(string, "Type: %s", cavalry_piece->type);
        Dlg.m_stats_type = string;

        // fill in health
        sprintf(string, "Defense: %d units", cavalry_piece->defense);
        Dlg.m_stats_defense = string;

        // fill in strength
        sprintf(string, "Offense: %d units", cavalry_piece->offense);
        Dlg.m_stats_offense = string;

        int ret = Dlg.DoModal();
    }
}
```

At the top of the netwavw.cpp file, add the code:

```
#include "ppstats.h"
```

The code to this point can be found in the directory /newar2/chapt5/e. Build and execute the application. Right double-click on the cavalry playing piece and take note of the dialog box that appears. With the dialog box in mind, we will go over the code just added to the application.

You will recall the ClassWizard created a couple of files for us when we added the playing piece statistics class to our application. One of the files added was ppstats.h. This file includes the complete definition of our class. We will be calling on our new class from the right double-click handler in the view object's code. This means the view object needs to know about the playing piece statistics dialog box class. By putting an *#include* statement followed by the ppstats.h name in the view object's netwavw.cpp code, we will be able to reference this class when we need to and the view object will know about it.

Some of the code in the right double-click handler should look very familiar. It is the code for determining whether or not the right double-click by the user occurred while the mouse cursor was on a playing piece. If the click was on a playing piece, we need to display our dialog box. Notice the reference to our stats class in the declaration section of this function. There is an entry called

```
PPSTATS Dlg;
```

This entry automatically creates a dialog box object of the PPSTATS class when the function is called. The object has been created, but the dialog box is not displayed on the screen. We have to do this manually.

Changing the About Box

The last thing we are going to do in our application is change the About box. AppWizard automatically creates one for us and includes generic information about the application. To change the dialog box, just start AppStudio and select Dialog Types. Double-click on the IDD_ABOUT entry. Use the tools in AppStudio to change the box. Save and close AppStudio when finished. You will need to build the project before your changes will be visible.

Chapter 6

Game Operations Support

The game foundation presented in the previous chapter gives us all of the hooks we need to create the remainder of the game. Two of the most important parts of the game are the operations window and the Buy Equipment dialog box. In this chapter, we will document the creation of these two entities.

Operations Window

The purpose of the operations window is to provide a panel of controls that the user can use in the play of the game. The window itself will be located to the right of the main window and is shown in Figure 6.1.

It will be created as a modeless dialog box, allowing it to co-exist with the main window and giving it the ability to be moved anywhere on the screen. The controls that will be created in the window are:

- armies-in-waiting list box
- current gold count
- war token indicator
- WAR button
- I Quit button
- Message button
- message list box

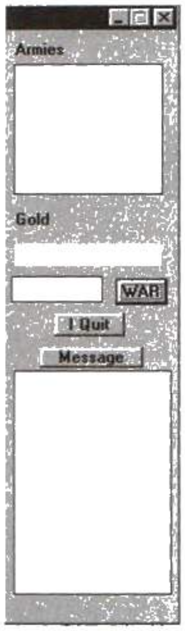


Figure 6.1
The Operations window.

The *armies-in-waiting list box* will display a list of armies that a player has purchased during game play. Instead of all of a player's armies simply being put on the playing field when they are purchased, the playing pieces will be listed in the armies-in-waiting list box by their type name. When ready to deploy the army, the player will double-click on any available army. The sprite representing the playing piece will be displayed on the playing field. An "army deployed" tag will be displayed in the armies-in-waiting list box at the location the player clicked.

The *current gold count* control will be a running count of the total gold coins the user has available. Gold coins will be necessary to purchase equipment and move in the game.

The *war token indicator* is a toggled static text control that indicates the user can start a war or must continue to wait. The war token will be circulated to all players in the game.

The WAR button is clicked by a user who has double-clicked on two armies that will to engage each other in war. The WAR button will only be effective if the appropriate armies have been selected.

The I Quit button is clicked when a user is giving up the fight. A click of this control will zero out the player's gold coins and remove their armies from the armies-in-waiting list box.

The Message button is clicked when a user wants to send a message to all players or one player in particular. Messages from other users as well as the game's control system are displayed in the Message list box.

Creating the Operations Window

The operations window will be created like any other dialog box or window by using the AppStudio application. With Visual WorkBench executing on your system and the NETWAR2 project loaded, click on Tools and AppStudio.

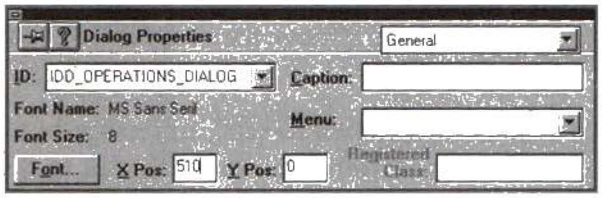
Select the Dialog Box Type and click New. Click OK in the New Resource dialog box that appears. You should have a blank dialog box ready to be manipulated. Let's create the operations window.

Step 1: Double-click on the title area of the new dialog box. In the properties dialog box that appears, change the ID to read "IDD_OPERATIONS_DIALOG". At the bottom of the properties dialog box, you will see two edit controls for the X and Y positions of the dialog box. In the X edit box, enter the value 510. The dialog box will now be created with its upper-left corner at 510, 0. You can blank out the caption edit control if you want as it will not be used. The properties box should be configured as shown in Figure 6.2.

Step 2: In the new dialog box, click on the OK button and press the Delete key. Do the same for the Cancel button.

Step 3: Resize the window surrounding the dialog box so that it takes up the majority of the screen vertically. Now resize the dialog box we are working on so that it is approximately 74 by 277. The size of the control you are currently using shows up in the lower-right corner of the AppStudio window. You don't need these numbers exactly, but you should be close. Your new dialog box should appear as shown in Figure 6.3.

Step 4: Before we start adding controls, we need to set the style of the dialog box. Double-click on the title bar of the dialog box again. When the properties box appears, you will see a control in the upper-right cor-



General

ID: IDD_OPERATIONS_DIALOG

Caption:

Font Name: MS Sans Serif

Font Size: 8

Menu:

X Pos: 510

Y Pos: 0

Registered Class:

Figure 6.2 *The properties box.*

ner. The current selection should read General. Click on the button and select Styles. This option will display a completely new set of controls in the Properties dialog box. Set all of the controls appropriately so that your properties match Figure 6.4.

Step 5: Now we are ready to start adding some controls. We will begin with the armies control. In the tools dialog box, click on the Text tool. Click somewhere in your new dialog box and drag a small box. Double-

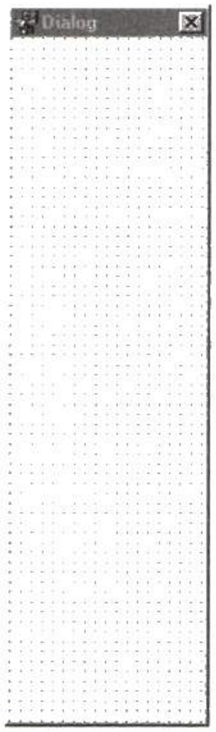
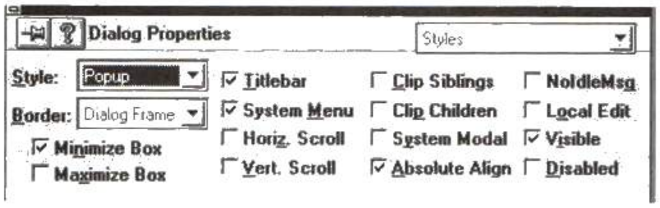


Figure 6.3
New dialog box.



Style: Popup
Border: Dialog Frame
☑ Minimize Box
☐ Maximize Box
☑ Titlebar
☑ System Menu
☐ Horiz. Scroll
☐ Vert. Scroll
☐ Clip Siblings
☐ Clip Children
☐ System Modal
☑ Absolute Align
☐ NoIdleMsg
☐ Local Edit
☑ Visible
☐ Disabled

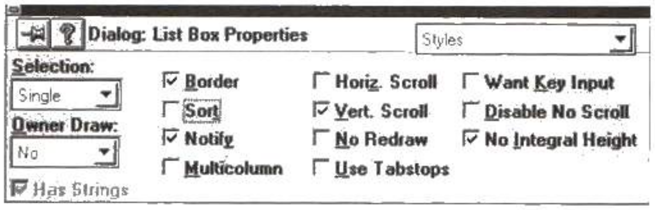
Figure 6.4 *Match your properties.*

click on the box created. In the Caption edit line, enter the name "Armies" and press return. Now resize the static control so the box just encloses the word "Armies". Move the text close to the top left of the dialog box.

Step 6: Go back to the tools dialog box and select the list box control. This is the control in the second column and five controls from the top. Position the cursor just under the "A" in "Armies". Drag a list box that is 64 by 60. Now double-click on the list box to bring up its properties box. The ID for this list box is "IDC_AVAILABLE_ARMIES". Change to the styles properties and make sure it looks like Figure 6.5.

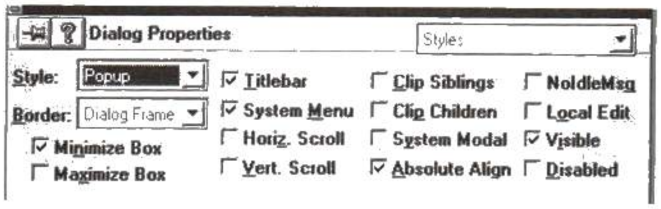
That's all we need to do to add the army list box control. Well, almost. After adding the remainder of the controls, we will have to use ClassWizard to relate the controls to our application. Your dialog box should now look like Figure 6.6.

Step 7: Using the text tool, add a static text with the word "Gold" in it. Position this text under the army list box and aligned with the text "Armies".



Selection: Single
Owner Draw: No
☑ Has Strings
☑ Border
☐ Sort
☑ Notify
☐ Multicolumn
☐ Horiz. Scroll
☑ Vert. Scroll
☐ No Redraw
☐ Use Tabstops
☐ Want Key Input
☐ Disable No Scroll
☑ No Integral Height

Figure 6.5 *Change the styles properties.*



Dialog Properties — Styles

Style: Popup

Border: Dialog Frame

[✓] Minimize Box
[ ] Maximize Box
[✓] Titlebar
[✓] System Menu
[ ] Horiz. Scroll
[ ] Vert. Scroll
[ ] Clip Siblings
[ ] Clip Children
[ ] System Modal
[✓] Absolute Align
[ ] NoIdleMsg
[ ] Local Edit
[✓] Visible
[ ] Disabled

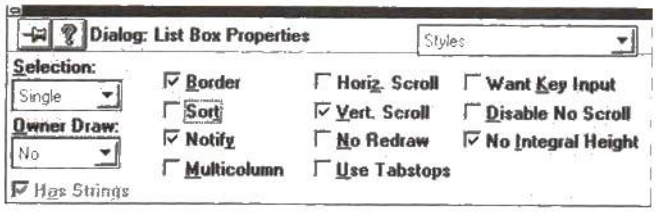
Figure 6.4 *Match your properties.*

click on the box created. In the Caption edit line, enter the name "Armies" and press return. Now resize the static control so the box just encloses the word "Armies". Move the text close to the top left of the dialog box.

Step 6: Go back to the tools dialog box and select the list box control. This is the control in the second column and five controls from the top. Position the cursor just under the "A" in "Armies". Drag a list box that is 64 by 60. Now double-click on the list box to bring up its properties box. The ID for this list box is "IDC_AVAILABLE_ARMIES". Change to the styles properties and make sure it looks like Figure 6.5.

That's all we need to do to add the army list box control. Well, almost. After adding the remainder of the controls, we will have to use ClassWizard to relate the controls to our application. Your dialog box should now look like Figure 6.6.

Step 7: Using the text tool, add a static text with the word "Gold" in it. Position this text under the army list box and aligned with the text "Armies".



Dialog: List Box Properties — Styles

Selection: Single

Owner Draw: No

[✓] Has Strings
[✓] Border
[ ] Sort
[✓] Notify
[ ] Multicolumn
[ ] Horiz. Scroll
[✓] Vert. Scroll
[ ] No Redraw
[ ] Use Tabstops
[ ] Want Key Input
[ ] Disable No Scroll
[✓] No Integral Height

Figure 6.5 *Change the styles properties.*

Step 8: Select the Edit control tool, which is in the second column and two from the top. Create an edit control under the "Gold" text that is 68 by 12. Double-click on the control to bring up the properties dialog box. The ID of this control is "IDC_OPERATIONS_AVAILABLE_GOLD". Change to the styles and click on Read Only and Border. The Read Only box should be checked and Border should not.

Our operations dialog box now looks like Figure 6.7.

Step 9: Select the Edit control tool again and make a control that is 40 by 12 and right aligned with the gold control just added. The ID for this control is "IDC_OPERATIONS_WAR_STATEMENT". You can leave the border on this one. Just select the Read Only option so that it is checked.

Step 10: Now select the button control. This control is in the second column and third from the top. Add a button to the left of the edit control just added. This button should be 24 by 12. Double-click on the button and enter the ID as "IDC_WAR_BUTTON". The caption for this button

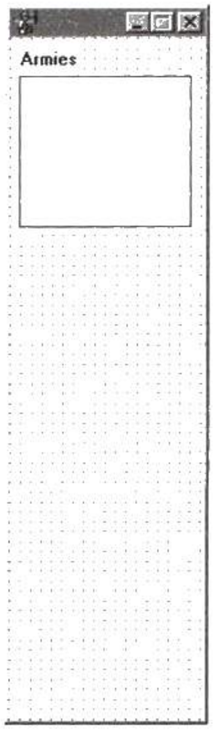


Figure 6.6
What the dialog box looks like.

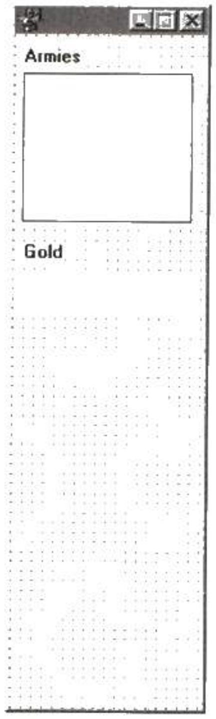


Figure 6.7
The new Operations dialog box.

should be "WAR". The operations dialog box now looks like Figure 6.8.

Step 11: Using the button control, center a button under the war controls that is 32 by 12.The ID for this button is "IDC_OPERATIONS_I_QUIT". The caption should read "I Quit".

Step 12: Again using the button control, add another button centered under the I Quit button that is 46 by 12. The ID for the button is "IDC_OPERATIONS_MESSAGE_BUTTON" and the caption is "Message".

Step 13: Last, select the list box control and add a list box under the message button that is 68 by 108. The ID for the list box is "IDC_OPERATIONS_MESSAGE_BOX". The Properties Styles should be equivalent to Figure 6.9.

The operations dialog box should now look like the one shown at the beginning of the chapter. Creating how the dialog box appears on the screen is only half of the story.

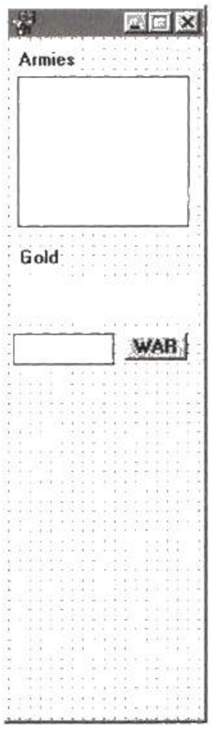


Figure 6.8
An updated Operations dialog box.

ClassWizard

Now we need to go to ClassWizard to interface the dialog box with our application. Click on Resources and ClassWizard. The first thing that appears is a dialog box asking us for the name of our new dialog box class. Enter "operations" and click on Create Class. The system will create the class and send us to the Message Map wizard.

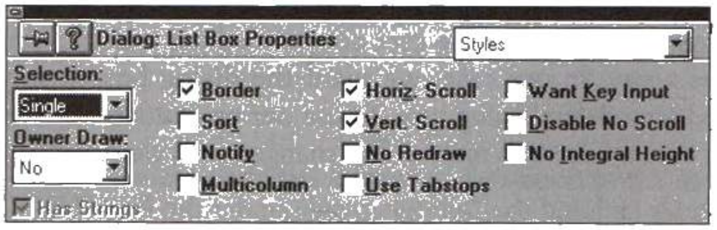
Under Message Map, we need to create message handlers for each of the buttons on our dialog box. The message handlers will be created in the code for the operations dialog box class. The file is called operatio.cpp. We will be using this file shortly. We have three buttons in our dialog box. Find IDC_WAR_BUTTON in the list of IDs and click on it. You will see a list of possible operations in the list box to the right of the ID list. Double-click on BN_CLICKED. A dialog box will appear asking about the member variable name. Just click on OK. This dialog box allows you to change the name of the message handler.

Find the ID for the I Quit button and the Message button. In both cases, select the BN_CLICKED message and let the system create the message handlers. When you are done, the ClassWizard window should appear as Figure 6.10.

Now we have to relate the gold coins count edit control we put on the operations dialog box to our application, so click on the Member Variables tab in the ClassWizard dialog box. Double-click on the ID called IDC_OPERATIONS_AVAILABLE_GOLD. A dialog box will appear asking for the name of the variable to relate to this control. Enter the name "m_operations_available_gold".

Do the same thing for the IDC_OPERATIONS_WAR_STATEMENT edit control. Name this control "m_operations_war_statement".

If you look at the list of controls, you will see that we haven't done anything with the list boxes. In the case of the message list box, we will only be adding things to the list box so we can handle all of the details in the operations class. There is something we can do with the Armies list box, however. Click on the Message Map tab in the ClassWizard dialog box. Find the ID called IDC_OPERATIONS_AVAILABLE_ARMIES and click on it. It should now be highlighted. In the box to the right, you will see all of the messages available for this list box. Recall that we said players will double-click on the armies that they wish to be placed on the playing field. Look in the list of messages and see if there is something that would relate to a double-click. The message is called LBN_DBLCLK. Double-click on this message and add a message handler. This message handler will be added to the operations class that represents this dialog box. We are finished with ClassWizard, so click on OK. Now click on the Save button and exit AppStudio.


Dialog: List Box Properties

Styles

Selection: Single

Owner Draw: No

✓ Has Strings

✓ Border

Sort

Notify

Multicolumn

✓ Horiz. Scroll

✓ Vert. Scroll

No Redraw

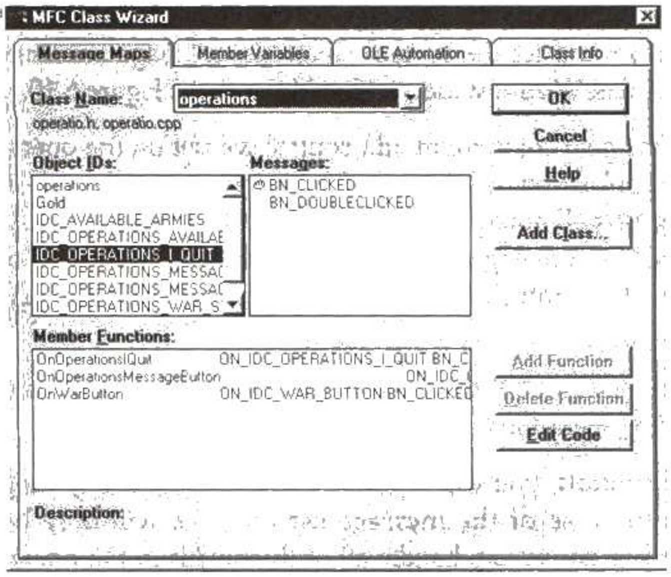
Use Tabstops

Want Key Input

Disable No Scroll

No Integral Height

Figure 6.9 *The properties styles.*



Class Name: operations

operatio.h, operatio.cpp

Object IDs:

```
operations
Gold
IDC_AVAILABLE_ARMIES
IDC_OPERATIONS_AVAILAE
IDC_OPERATIONS_I_QUIT
IDC_OPERATIONS_MESSAC
IDC_OPERATIONS_MESSAC
IDC_OPERATIONS_WAR_S
```

Messages:

```
BN_CLICKED
BN_DOUBLECLICKED
```

Member Functions:

```
OnOperationsIQuit             ON_IDC_OPERATIONS_I_QUIT BN_C
OnOperationsMessageButton                             ON_IDC_
OnWarButton                   ON_IDC_WAR_BUTTON BN_CLICKED
```

Figure 6.10 *The ClassWizard window.*

Displaying the Operations Dialog Box

The operations dialog box is finished and the project can be recompiled, so the operations dialog box is included in our applications executable file. There is a slight problem, however. We haven't added anything to our main object, so the operations dialog box isn't displayed on the screen.

If we think about displaying the operations dialog box, we find that we want the window to be displayed as soon as the application is started. This is similar to an initialization task, so we should start the operations dialog box in the view's constructor.

But wait a minute . . . . The operations dialog box is a dialog box. Normally, a dialog box is displayed on the screen, the user does something with the dialog box, and it goes away. In our case, we want the dialog box to stay around. This is called a modeless dialog box. To use our operations window as a modeless dialog box, we have to make a few changes to its class.

Open the file operatio.h and add the following code under the first PUBLIC heading.

```
operations(CView* pView);  // standard constructor
BOOL Create();
```

This code tells the class that we are creating a new constructor that takes as its parameter a view object. Since our application is creating the operations window from a view object, this should seem appropriate. Now this isn't something that we are creating. This secondary constructor is available to all CDialog class objects. The second statement is declaring a function called Create() that will override the dialog's standard create function.

The actual code for these new methods has to be added to the source file for our dialog class. Open the file operatio.cpp and add the following code.

```
operations::operations(CView* pView) // modeless constructor
   : CDialog()
{
 m_operations_available_gold = "";
 m_operations_war_statement = "";
}

BOOL operations::Create()
{
 return CDialog::Create(operations::IDD);
}
```

The first function, the constructor, is modeled after the standard constructor for a dialog box. Notice that the only code in the function is an initialization of the two-member variable we defined with ClassWizard. The CDialog class knows that when a view object creates a dialog box using this constructor, a modeless dialog box is being created. The second function is the Create() method override.

All of this new code sets the operations dialog class for modeless operation. Now we need to add code to our application's view object to actually create the operations window. Start by opening the file netwavw.h. Add the following statement to the top of the file just before the class definition for our view:

```
class operations;
```

This statement performs a forward definition of the operations class, which is the class for our operations dialog box. The actual definition of the class will be added to our application in the source file for the view object. In the same file, add the following statement in the our view's class definition under the first PUBLIC heading:

```
operations* ops_dlg;
```

This statement declares an object pointer of type *operations*. The variable *ops_dlg* will point to our operations dialog box. Save and close this file.

Now open the file netwavw.cpp and add the following statement at the top of the file before any of the methods:

```
#include "operatio.h"
```

This statement includes the header file of our operations class. The header file includes the definition of the class. Next, add the following code at the end of the view constructor method:

```
//create operations dialog box
 ops_dlg = new operations(this);
 if (ops_dlg->GetSafeHwnd() == 0)
  ops_dlg->Create();
```

The code just added is where the operations dialog box is called by our view. The first statement creates a new *operations* object and relates it to the *ops_dlg* variable. Notice the *this* parameter sent to the constructor. By using the *this* parameter, the operations class invokes the modeless constructor we added to the operations class. Next, the code checks to make sure that the dialog pointer returned by the operations class constructor is valid and then calls the Create() method to display the dialog box.

To see the complete application, rebuild and execute the project. You will see the operations dialog box appear at the right part of the screen. You can move the dialog box anywhere on the screen. The code to this point can be found in the directory /netwar2/chapt6/a.

Armies

Our game foundation only creates a single playing piece. This single piece was used as a demonstration to show how to create sprites. In the game, players will be purchasing and displaying many different playing pieces. The armies-in-waiting list box will be used to keep track of the playing pieces.

In the final game, all of the playing pieces purchased using the Buy Equipment dialog box will be automatically placed in the armies-in-waiting list box. Since the Buy Equipment dialog box hasn't been created yet, we will place playing pieces in the list box manually.

In normal dialog boxes, the list boxes are initialized when the dialog box is constructed. In our case, the operations dialog box won't contain anything when it is initialized and displayed. As the players purchase playing pieces, the pieces are added to the army list box dynamically. This could pose a problem, since our view object doesn't control the army list box in the operations dialog box. Only the operations dialog code can add things to the operations dialog box controls. We have to come up with some way of letting the operations dialog box know that we, the view object, want something added to the army list box. The way to do this is using a message. All of the windows in Windows can send and receive messages. We could send a message to the operations dialog box telling it to add an army to the list box.

Adding an Army to the List Box

A playing piece has been created, and we want to get its name put in the list box control on the operations window. The playing piece should not be displayed on the playing field, since it has only been purchased and not placed. The view object in our application has complete control over the operations window and can in fact use variables that are defined in the operation window's class. What we can do is place a member variable in the operations class that will be used to hold the name of the playing piece to be put in the list box. The view object can send a message to the operation window telling it to put the name in the member variable in the list box.

Open the operatio.h file and add the following code to the first PUBLIC heading:

```
int total_armies;
char entries[20];
```

The first variable added is a counter for the total number of armies that have been placed in the list box. This variable needs to be initialized, so open the file operatio.cpp and place the code

```
total_armies = 0;
```

in both of the constructors.

The second variable is a character string that will be used to hold the name of the playing piece to put in the list box.

In the operatio.h file, add the line

```
#define WM_OPERATIONS_ADD_ARMIES       WM_USER + 100
```

at the very top.

This code creates a defined constant that relates to the value WM_USER+100. This is the message that our view object will send to the operations window when the army should be added to the list box. The constant WM_USER is the first message that Windows allows user applications to use.

When the view object sends this message, the operations window class needs to respond to it. A message handler would be appropriate. In the operatio.h file, a heading called PROTECTED can be found at the end of the class definition. Add the following line of code just before the DECLARE_MESSAGE_MAP() line:

```
long OnAddArmies(UINT wParam, LONG lParam);
```

This is a definition of the message handler method for the ADD_ARMIES message.

Close the operatio.h file and move to the operatio.cpp file. Add the following function in this file:

```
long operations::OnAddArmies(UINT wParam, LONG lParam)
{
 CListBox* box;

 box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
 box->ResetContent();
 box->AddString(entries);

 return 1;
}
```

This is the message handler. It starts out by getting an object handle for the control IDC_AVAILABLE_ARMIES. Recall that this is the ID we gave to the Armies list box. Once a handle is available, the contents of the list box are cleared with the ResetContent() method. The name of our playing piece is added to the list box with the AddString() method. The only parameter to add to this method is the character string of the item.

Before we can build and execute the new application, we have to make two more changes. At the top of this file, you will find a message map declaration. The declaration begins with BEGIN_MESSAGE_MAP and ends with END_MESSAGE_MAP. Just after the BEGIN_MESSAGE_MAP statement, add the code

```
ON_MESSAGE(WM_OPERATIONS_ADD_ARMIES, OnAddArmies)
```

This statement says to the operations window that if the message WM_OPERATIONS_ADD_ARMIES is received, call the method OnAddArmies() to take care of the response.

Close this file and open the file netwavw.cpp. We have to add code to our view object so that the army is placed in the list box. Move to the constructor and find the code where our playing piece is initialized. In this code, you will find that the parameter relating to the ACTIVE member variable for the playing pieces is sent as TRUE. Change this parameter to FALSE. Since the playing piece is not put on the playing field, we don't want the piece to be active.

We are going to add some temporary code next. After the definition and creation of the operations window, add the code

```
strcpy(ops_dlg->entries, "cavalry");
ops_dlg->SendMessage(WM_OPERATIONS_ADD_ARMIES, 0, 0L);
```

The first line of code copies the name "cavalry" to the member variable we defined in the operations window class. The second statement sends the message WM_OPERATIONS_ADD_ARMIES to the operations window.

Move to the OnDraw() method and replace the code

```
  if (cavalry_piece->show)
   cavalry_piece->Redraw(pDC);
  else
```

```
    {
     cavalry_piece->Start(pDC, 100, 100);
     cavalry_piece->show = TRUE;
    }
```

with the code

```
if (cavalry_piece->active == TRUE)
  {
    if (cavalry_piece->show)
     cavalry_piece->Redraw(pDC);
    else
    {
     cavalry_piece->Start(pDC, 100, 100);
     cavalry piece->show = TRUE;
    }
  }
```

Close this file, and build and execute the new code. You will find the "cavalry" name in the list box on the operations window (Figure 6.11).

Now we need to get the playing piece from the list box to the playing field. All of the code to this point can be found in the directory /netwar/chapt6/b.

Removing an Army from the List Box

A good portion of the code we need to get our playing piece from the Armies list box to the playing field has already been provided for us by Visual C++ and the MFC library. Recall that when we designed the operations window, we accessed ClassWizard and included a handler for a double-click in the Armies list box. This handler method is where we will start.

Open the operatio.cpp file and find the method called OnDblclkAvailableArmies(). Add this code to the function

```
CListBox* box;
  int temp;

  box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
  temp = box->GetCurSel();

  if (list_box[temp] != -1)
  {
    selected_army = list_box[temp];
    list_box[temp] = -1;
```

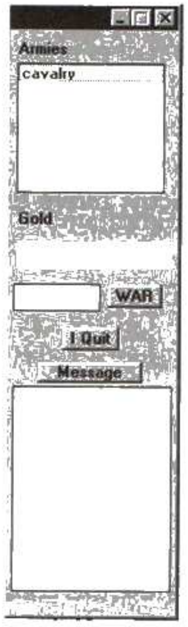


Figure 6.11
Cavalry name is found in the list box.

```
    box->DeleteString(temp);
    box->InsertString(temp, "Army Deployed");

    if (m_pView != NULL)
     m_pView->SendMessage(WM_ARMY_SELECTED, IDOK);
  }
```

This is all of the code necessary to select an army from the list box and prepare it to be placed on the playing field. The code begins by obtaining an object handler to our list box. Once we have the handler, a call is made to the method GetCurSel(). This method returns an index number representing the place in the list box where the mouse was double-clicked. The value returned is from 0 to one less than the number of entries in the list box. In our case, there is only one entry in the list box, but we have to be prepared for multiple armies in the list box.

After the index of the selected army is obtained, a check is made of the variable array *list_box*. This variable won't be familiar because we haven't added it yet. Open the operatio.h file and add the statement

```
int list_box[5];
```

under the first PUBLIC heading. This variable will be used to indicate whether or not an army has already been deployed. Obviously, if a player double-clicks on an army that is already on the playing field, the playing piece shouldn't be duplicated. A value of 1 indicates that the playing piece has already been selected. All other values indicate that the army is available for deployment. In order for this variable array to work correctly, move to the OnAddArmies() method in the operatio.cpp file and add the following code to the end of the method.

```
list_box[total_armies] = 0;
total_armies++;
```

This code will place the value of 0, indicating an available army, at the next available position. This position is represented by the value in the *total_armies* variable. Once the array position is filled, the *total_armies* variable is incremented by one.

Now back to our selection code. After a look in the *list_box* array, the code will know whether or not the playing piece has already been selected. If the playing pieces has been selected, nothing happens. Otherwise, the next two pieces of code set a PUBLIC member variable *selected_army* to the value in the *list_box* array. In our current state this statement is not necessary; however, in the future, the *list_box* array will contain the playing piece number from the view object. As playing pieces are created by the view object and placed in the list box, it will record the playing piece's number in the *list_box* array.

The PUBLIC member variable has to be defined in the operatio.h file under the first PUBLIC heading. The code is

```
int selected_army;
```

A value of 1 is placed in the playing piece's position in the *list_box* array to indicate that this playing piece has been selected. Since the playing piece has been deployed, we should no longer leave its name in the Armies list box. The member method DeleteString() removes the string entries at the position given as a parameter. The function InsertString() inserts a string entry at the position given as a parameter. These two functions will remove the name of the playing piece and reinsert the string "Army Deployed" in the appropriate position. Now the player can determine at a glance which armies have been deployed and which are available.

The last two lines of code check to make sure that the operations window was created by a view object and then send a message called WM_ARMY_SELECTED to the view object to indicate that a player has selected an army for deployment on the playing field. The view object will be responsible for responding to this message.

In determining whether or not the operations window was created by a view object, a new variable called *m_pView* was used. This variable is defined in the operatio.h file under the first PUBLIC heading as

```
CView* m_pView;
```

It is associated with a value in both of the constructors found in the operatio.cpp file. In the first constructor designed for modal dialog boxes, add the line

```
m_pView = NULL;
```

In the constructor designed for modeless dialog boxes, add the line

```
m_pView = pView;
```

Now if the operations window is created in a standard dialog box, the *m_pView* variable will have a value of NULL and will not allow a message to be sent to the view object. Otherwise, this variable will contain an object handle to our view object and the message will be sent.

The message WM_ARMY_SELECTED has to be defined so that our operations window class and the view class know about it. The place to put this type of information is in the netwar2.h file. This is the main header file for all code files in our project. Open this file and place the following line at the top

```
#define WM_ARMY_SELECTED WM_USER+10
```

Setting up the View Object

To this point, our operations window knows and responds to a double-click in the Armies list box. The handler does all of the necessary housekeeping work, sets the playing piece value in the PUBLIC member variable *selected_army*, and sends a message to the view object.

The view object must respond to this message. The message handler is going to be called OnArmySelected(). Open the netwavw.h file and replace the code

```
afx_msg void OnRButtonDblClk(UINT nFlags, CPoint point);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

with

```
afx_msg void OnRButtonDblClk(UINT nFlags, CPoint point);
//}}AFX_MSG
long OnArmySelected(UINT wParam, LONG lParam);
DECLARE_MESSAGE_MAP()
```

All we have done is define a new member method for our view class. Now open the file netwavw.cpp and add a message handler to the message map macro by replacing the code

```
BEGIN_MESSAGE_MAP(CNetwar2View, CView)
      //{{AFX_MSG_MAP(CNetwar2View)
      ON_WM_LBUTTONDOWN()
      ON_WM_MOUSEMOVE()
      ON_WM_LBUTTONDBLCLK()
      ON_WM_LBUTTONUP()
      ON_WM_RBUTTONDBLCLK()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

with

```
BEGIN_MESSAGE_MAP(CNetwar2View, CView)
      ON_MESSAGE(WM_ARMY_SELECTED, OnArmySelected)
      //{{AFX_MSG_MAP(CNetwar2View)
      ON_WM_LBUTTONDOWN()
      ON_WM_MOUSEMOVE()
      ON_WM_LBUTTONDBLCLK()
      ON_WM_LBUTTONUP()
      ON_WM_RBUTTONDBLCLK()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Notice that our message handler statement has been placed outside of the AFX code. All of the AFX code is designed for the ClassWizard application. Since the ClassWizard is not creating the code that we are adding, we have to make sure that we put our code outside of the AFX information.

The message handler has been defined in the class definition for our view, and a message map macro has been created that sends all WM_ARMY_SELECTED messages to the handler OnArmySelected(). All we need now is the message handler. Move to the bottom of the netwavw.cpp file and add the code

```
long CNetwar2View::OnArmySelected(UINT wParam, LONG lParam)
{

 cavalry_piece->active = TRUE;
 Invalidate(TRUE);

 return 1;
}
```

This message handler is only called when an army is selected from the operations window Armies list box. Since we only have one playing piece, we know that it was the piece selected, so its *active* member variable is set to TRUE and a call is made to the method Invalidate(). This method causes a WM_PAINT message to be sent to the application and the OnDraw() method is executed. Since we have set the *active* variable to TRUE for our playing piece, it will be displayed on the screen.

Later, we will see that this function has to look at the *selected_army* variable in order to determine which of the player's playing pieces has been activated. We will add this code when we design the Buy Equipment dialog box later in this chapter.

Save and close all of the open files, and build and execute the application. You can now double-click on the cavalry name in the Armies list box and the playing piece will appear on the playing field (Figure 6.12). The code up to this point is in the directory /netwar/chapt6/c.

Current Gold Count

Each of the players in our game will have a starting amount of gold coins that can be used to purchase playing pieces and to move them. After many different test games, we found that 500,000 coins was appropriate, given the purchase price of our playing pieces.

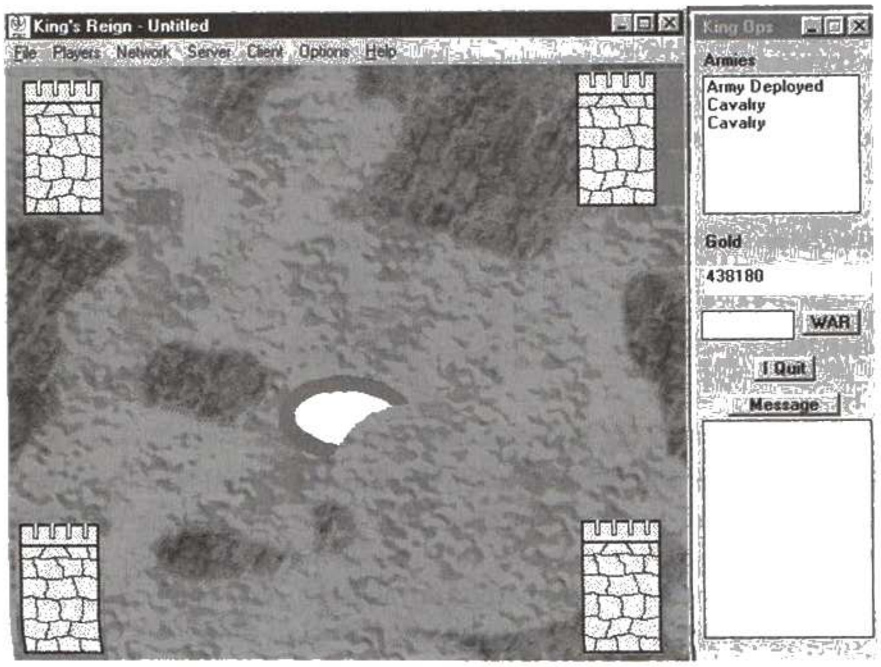


Figure 6.12 *Click on the cavalry name, then the playing piece will appear.*

In designing the game, every attempt is made to create only one executable program per player. It would be bad design if every player had to have a different version of the game. With this in mind, we are going to create a player class in which we can place player-specific information such as total gold coins. At the same time, there is going to be a variable called *who_am_i* that is defined as a view object public member. This variable will be used to identify which player in the game we are supposed to be. First we should define the player class.

The Player Class

To create the player class, a new file is necessary. In the Visual WorkBench, click on File and New. A window will appear. Type in the following code:

```
class CPlayers
{
public:         //variables
   long gold_coins;
public:         //methods
```

```
    Cplayer();
    ~CPlayer();
};
```

Save this file as player.h. Click on File and New again to get another blank file. In this file, type the code

```
#include "player.h"
#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
#include "netwavw.h"

CPlayers::CPlayers()
{
 gold_coins = 500000;
}

CPlayers::~CPlayers()
{
}
```

Save this file as player.cpp. In the source file, we have two methods; a constructor and a destructor. The constructor only does one thing at this point, and that is to initialize the number of gold coins to 500,000.

Now that the player class is defined, we can put it to use. Open the file netwarvw.h and add under the first PUBLIC heading the lines

```
int who am i;
Cplayer* players[4];
```

At the top of this file, add the following line of code:

```
#include "player.h"
```

Open the netwarvw.cpp file and add the following code at the bottom of the constructor:

```
who am_i = 0;
players[who_am_i] = new CPlayer;
```

This is all we need for the application at the moment. In the later versions of the game, each player is assumed to be player 0, the server, until the players say otherwise. By using this type of setup, the game can be "played" by a single player.

In order for the project file to compile our new class, click on Project and then Edit. Double-click on the file name player.cpp and then close the displayed dialog box. The system will automatically add the new file and its class to the project file.

Now we have to get to the task at hand, which is to display the current number of gold coins in the operations window. Displaying the gold coins in the operations window will be performed in the same manner as in the Armies list box. A variable will be declared in the operations window class where the number of coins to display will be copied by the view object. Once copied, a message, WM_OPERATIONS_UPDATE_GOLD, is sent to the operations window. A message handler will take the gold coins value and display it in the edit line we put on the window.

Let's begin by adding the appropriate code to the operations class. Open the operatio.h file and add the definition of the message that will be used to indicate a gold update needs to take place.

```
#define WM_OPERATIONS_UPDATE_GOLD WM_USER + 101
```

Next, add the following code to the class in the first PUBLIC area:

```
long total_gold;
```

Move down to the PROTECTED heading and add a message handler function definition

```
long OnUpdateGold(UINT wParam, LONG lParam);
```

This is all that we need in the header file. Open the operatio.cpp file and add the actual message handler function

```
long operations::OnUpdateGold(UINT wParam, LONG lParam)
{
   CEdit* box;
   char buffer[15];

   box = (CEdit*)GetDlgItem(IDC_OPERATIONS_AVAILABLE_GOLD);
   sprintf(buffer, "%ld", total_gold);
   box->SetSel(0, -1, FALSE);
   box->ReplaceSel(buffer);

   return 1;
}
```

Move up to the message map section and add the code

```
ON_MESSAGE(WM_OPERATIONS_UPDATE_GOLD, OnUpdateGold)
```

Look back at the message handler function. As in the other cases, we have to get the object handle to the edit line for the gold control. The gold control is an edit box of the object type CEdit. After the message handler is obtained, the message handler takes the value indicating the number of gold pieces for the player and converts this value to a string. The string is placed in the character string variable *buffer*. The next line of code calls the CEdit method SetSel(). This function selects some part of an edit control. In our code, we have passed the parameters 0, 1, FALSE. This specific pattern of parameters tells the SetSel() method to select all of the text in the edit control. The last code statement calls the method ReplaceSel() with our buffer parameter. This method replaces the currently selected text in the edit control box with the text given as a parameter.

For our purposes, this code works just fine. All of the text in the gold edit control is replaced with the current number of gold coins. As fast as we can put a new value in the operations *total_gold* variable and send a message to the operations window, the text in the operations window will change.

Open the file netwarvw.cpp and add the following code to the end of the constructor.

```
ops_dlg->total_gold = players[0]->gold_coins;
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, 0, 0L);
```

Save and close all of the files, and build and execute the program to see the gold coins appear in the operations window as shown in Figure 6.13.

War

When a player has highlighted one of his or her playing pieces and that of another player, they can begin a battle. The battle is initiated with a click of the WAR button. When we were designing the operations window, the ClassWizard application was used to create a message handler for the WAR button. The message handler skeleton method can be found in the operatio.cpp file. Open this file and find the function void operations::OnWarButton(). Add the following code to this method:

```
if (m_pView != NULL)
{
 m_pView->SendMessage(WM_WAR_BUTTON, IDOK);
}
```

There really isn't much to the message handler. When the user clicks on the WAR button, the code checks to make sure that the view object was created through a modeless constructor and immediately sends a message to the view object, letting it know the button was clicked. That's it. Since most of the work for a war has to be performed by the view object, we might as well allow it to handle the button instead of copying information to the operations window.

Before we can use this code, however, we have to set up the message and its handler in the view object code. Since both the view object and the opera-

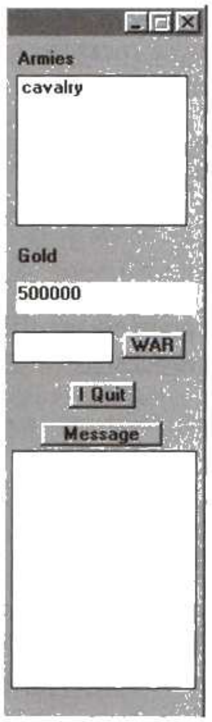


Figure 6.13
Gold coins.

tions window must know about the message, it will be defined in the netwar2.h main header file.

Open this file and add the code

```
#define WM_WAR_BUTTON WM USER + 11
```

under the other message definition found in the file.

Add a message handler prototype to the view class definition in the netwavw.h file. The code is

```
long OnWar(UINT wParam, LONG lParam);
```

Open the netwavw.cpp file and add the message handler to the end of the file.

```
long CNetwar2View::OnWar(UINT wParam, LONG lParam)
{
 MessageBox("Clicked War Button", "OK", MB_OK);

 return 1;
}
```

The code in the OnWar() method is just to let us know if everything is working correctly. The code for the war will be created in Chapter 7.

War Token Indicator

During the play of the game, only one player will be allowed to initiate a war at a time. To indicate to players that they can click the WAR button and start a war, a small edit line control is used to display one of two pieces of text; "Okay For" and "Not Okay". To accomplish this task, we are going to create two message handlers in the operations class. One of the handlers, called OnWarDo(), will display the "Okay For" message. The handler OnWarUndo() will display the "Not Okay" text.

The first step is to add the message handler prototypes to the operations class.

```
long OnWarUndo(UINT wParam, LONG lParam);
long OnWarDo(UINT wParam, LONG lParam);
```

The OnWarDo() method will be invoked when the message WM_OPERATIONS_ENABLE_WAR and OnWarUndo() is invoked with WM_OPERA-

TIONS_DISABLE_WAR. Add the message definitions at the top of the operations header file.

```
#define WM_OPERATIONS_ENABLE_WAR WM_USER + 102
#define WM_OPERATIONS_DISABLE_WAR WM_USER + 103
```

Now relate the message handlers to the messages in the message map macro at the top of the operations source file.

```
ON_MESSAGE(WM_OPERATIONS_ENABLE_WAR, OnWarDo)
ON_MESSAGE(WM_OPERATIONS_DISABLE_WAR, OnWarUndo)
```

Now it's time for the message handlers.

```
long operations::OnWarDo(UINT wParam, LONG lParam)
{
   CEdit* box;

   box = (CEdit*)GetDlgItem(IDC_OPERATIONS_WAR_STATEMENT);
   box->SetSel(0, -1, FALSE);
   box->ReplaceSel("Okay For");

   return 1;
}

long operations::OnWarUndo(UINT wParam, LONG lParam)
{
   CEdit* box;

   box = (CEdit*)GetDlgItem(IDC_OPERATIONS_WAR_STATEMENT);
   box->SetSel(0, -1, FALSE);
   box->ReplaceSel("Not Okay");

   return 1;
}
```

In both message handlers, the object handle for the war statement edit control on the operations window is obtained first. Next, all of the text in the edit control is selected and replaced with the appropriate text. That's all we have to do.

To test things out, add the code

```
ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
```

to the end of the view object constructor. Save, build, and execute the code. and you will see the "Okay For" text appear in the *war* statement edit control. See Figure 6.14.

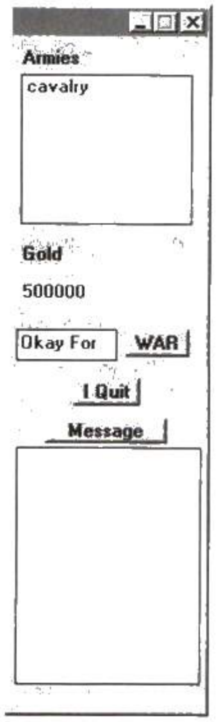


Figure 6.14
"Okay For" text appears in the war statement edit control.

I Quit

Much of the code put into the operations window is just support code for the view object. Since the operations window contains all of the controls for the game, we have to be able to let the view object know when one of the buttons is clicked. This is the case for the I Quit button. When this button is clicked, the user wishes to surrender. Since the I Quit button was put in the operations window in AppStudio, we were able to use ClassWizard to set up the message handler for the button automatically. What we need to do is set up the message and its handler in the view object manually. First, though, let's look at the message handler in the operations window.

Open the operatio.cpp file and fill in the function with the code

```
if (m_pView != NULL)
    m_pView->SendMessage(WM_QUIT_BUTTON, IDOK);
```

When the user clicks the I Quit button on the operations window, the message handler for the button sends the message WM_QUIT_BUTTON to the view object. Now we need to set up the view object information.

First, put the code

```
#define WM_QUIT_BUTTON WM_USER + 12
```

in the netwar2.h file.

Now put the message handler prototype in the view class (use the netwarvw.h file).

```
long OnQuitButton(UINT wParam, LONG lParam);
```

Add the handler to the source file

```
long CNetwar2View::OnQuitButton(UINT wParam, LONG lParam)
{
 MessageBox ("Clicked Quit Button", "OK", MB_OK);
 return 1;
}
```

and relate the message to the message handler in the message map macro section.

```
ON_MESSAGE(WM_QUIT_BUTTON, OnQuitButton)
```

Now save, build, and execute. When you click the Quit button, a dialog box will appear letting you know you clicked the Quit button as shown in Figure 6.15.

Messaging

When players are playing the game, they need to be made aware of certain things happening during game play. This could include the start of a war and its outcome as well as a player quitting the game. To accomplish this, a message list box is included with the operations window. In addition to system messages, each player will have the ability to send messages to a single player or all players. Here we will look at adding the code necessary for the game itself to send messages to the list box. In Chapter 12 we will add player messages.

Message List Box

The message list box is just a place where small messages can be displayed and scrolled through if necessary. The only operation that has to be provided for is adding a message to the list box and then displaying it. We will do this with a message and an associated handler. The message we will use is called WM_OPERATIONS_UPDATE_MESSAGE. Add the following code to the operatio.h file:

```
#define WM_OPERATIONS_UPDATE_MESSAGE WM_USER + 104
```

Move down to the definition of the operations window class and add the method declaration:

```
long OnUpdateMessage(UINT wParam, LONG lParam);
```

Open the operatio.cpp file and add a message macro to the message map such as

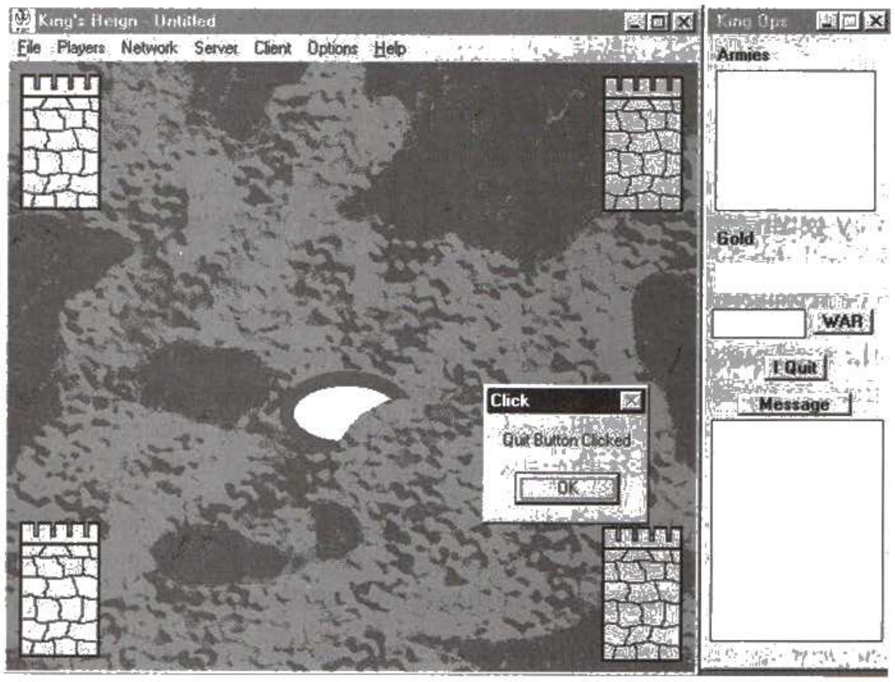


Figure 6.15 *Clicking on the quit button.*

```
ON_MESSAGE(WM_OPERATIONS_UPDATE_MESSAGE, OnUpdateMessage)
```

This macro will provide the link between the reception of the message and the message handler that will do the message response. The message handler is called OnUpdateMessage() and should be added to the end of the operatio.cpp file.

```
long OPERATIONSDIALOG::OnUpdateMessage(UINT wParam, LONG lParam)
{
   CListBox* box;

   box = (CListBox*)GetDlgItem(IDC_OPERATIONS_MESSAGE_BOX);

   box->SetHorizontalExtent(256);

   total_messages++;
   if (total_messages > 10)
   {
      box->DeleteString(0);
      box->AddString(new_message);
   }
   else
   {
      box->AddString(new_message);
   }

   return 1;
}
```

This is the code for adding a message to the message list box. The code begins by getting the handle to the list box itself. Next, it sets the total horizontal scroll distance. This allows for messages that are longer than the list box to be scrolled through.

Next, the total number of messages in the list box is incremented. If there are more than 10 messages in the list box, the very top message is removed and the new message is added to the end of the list of messages. If there are fewer than 10 messages in the box, the new message is just added to the list box.

One of the things we have to look at is this variable called *new_message*. This variable is an array character that the view object copies a message into before sending an update message to the operations window.

The *new_message* and the *total_messages* variables are defined in the operatio.h file. Before we add these variables, move up to the operations window constructor and add the code

```
total_messages =0 ;
```

This will zero out the *total_messages* variable so that the system knows that there are currently no messages in the list box.

In the operatio.h file, add the following declarations in the first PUBLIC area.

```
int total_messages;
char new_message[50];
```

Everything is now ready for messages to be placed in the box. To add a message, the view object would execute the code

```
strcpy(ops_dlg->new_message, "The message");
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
```

Once the message is received by the operations window, the message in the *new_message* variable is placed in the message list box. The code to this point can be found in the chapt6/d directory.

Buy Equipment Dialog Box

One of the most important parts of our game and its design is the Buy Equipment dialog box. It is this dialog box that will be used at the beginning of play and any time during play to purchase the playing pieces a player will use in the game. The dialog box appears as shown in Figure 6.16.

It is quite a large dialog box, but it only has one function: to purchase playing pieces. Four areas must be considered in designing the final dialog box: AppStudio design, ClassWizard controls, Interface to the view object, and the interface to the operations window.

AppStudio Design

To create the dialog box, we will use AppStudio. First, start the application. The properties for the dialog box should be as shown in Figure 6.17.

Using the illustration of the dialog box given earlier, add all of the static text controls. You don't need any additional information to add them other than the size of the dialog box, which is 358 by 266.

Once all of the text controls are added, the icon controls can be put in place. The icon controls are just small pictures that give an illustration of the playing piece available for purchase. There is really nothing special about them, but they do add graphically to the dialog box.

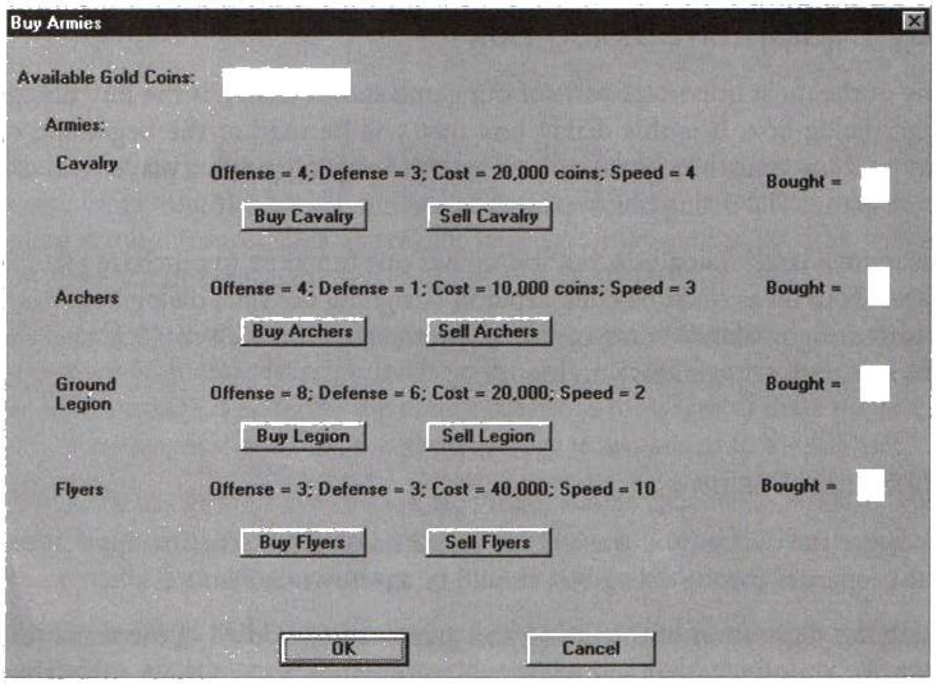
The first step in creating the icon controls is to design the icons. Click on the AppStudio dialog box that has all of the different resource types. Click on the Icon type and then New. Using the tools, create an icon that looks like a cavalry playing piece. Once you are finished with the icon, close the window. Click on the Properties button at the bottom of the dialog box. Change the ID to "IDI_CAVALRY".

Follow the same procedure for the three remaining types of playing pieces. The IDs for each of the icons are:

IDI_FLYER

IDI_ARCHER

IDI_LEGION


Buy Armies

Available Gold Coins:

Armies:

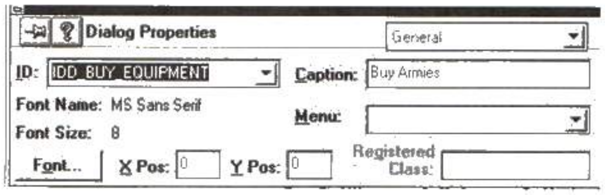
Cavalry Offense = 4; Defense = 3; Cost = 20,000 coins; Speed = 4 Bought =

Archers Offense = 4; Defense = 1; Cost = 10,000 coins; Speed = 3 Bought =

Ground Legion Offense = 8; Defense = 6; Cost = 20,000; Speed = 2 Bought =

Flyers Offense = 3; Defense = 3; Cost = 40,000; Speed = 10 Bought =

Figure 6.16 *Buy equipment dialog box.*



ID: IDD_BUY_EQUIPMENT
Caption: Buy Armies
Font Name: MS Sans Serif
Font Size: 8
Menu:
X Pos: 0
Y Pos: 0
Registered Class:

Figure 6.17 *Creating the dialog box using AppStudio.*

The icon controls are added using the bitmap tool. The bitmap tool is located on the second column of the tools dialog box and the first picture. Click on the tool. Now move to the dialog box and make a small box to the right of the static text "cavalry". Now double-click on the bitmap control that appears on the dialog box. Make the properties appear as shown in Figure 6.18.

Create three additional bitmap controls for each of the remaining playing pieces. Each of the Icon edit lines corresponds to the appropriate icon just designed.

The next step is to add the buttons for each of the playing pieces. Each playing piece has two buttons; one for buying and one for selling. The IDs for each button are:

Cavalry playing pieces

Buy button = IDC_BUY_CAVALRYS

Sell button = IDC_SELL_CAVALRYS

Legion playing pieces

Buy button = IDC_BUY_LEGIONS

Sell button = IDC_SELL_LEGIONS

Archer playing pieces

Buy button = IDC_BUY_ARCHERS

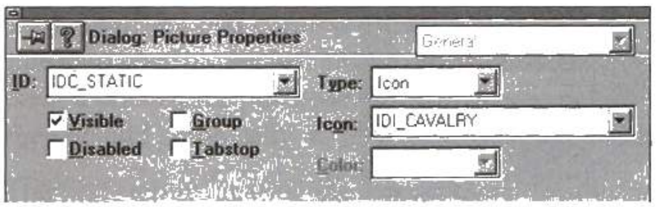
Sell button = IDC_BUY_ARCHERS

Flyer playing pieces

Buy button = IDC_BUY_FLYERS

Sell button = IDC_SELL_FLYERS

The last controls that we need to add to the Buy Equipment dialog box are the edit line controls that will display the number of playing pieces pur-



ID: IDC_STATIC
Type: Icon
☑ Visible ☐ Group
☐ Disabled ☐ Tabstop
Icon: IDI_CAVALRY
Color:

Figure 6.18 *The appropriate properties.*

chased. These are placed at the end of each playing piece section. The ID to use for each edit control is:

cavalry playing piece = IDC_BUY_CAVALRY_COUNT

legion playing piece = IDC_BUY_LEGION_COUNT

archer playing piece = IDC_BUY_ARCHER_COUNT

flyer playing piece = IDC_BUY_FLYER_COUNT

That's all of the controls for the Buy Equipment window.

ClassWizard Controls

All of the controls on the window don't do much good without message handlers and member variables to relate the controls to the dialog box class, just as in the case of the operations window. Close the dialog box that contains the buy dialog box we are creating. Now, before closing AppStudio, click on Resources and then ClassWizard. The ClassWizard will display a dialog box asking for the class name of the dialog box we are creating. Enter the name "buy_equipment" and click Create Class.

In the ClassWizard dialog box that appears, click on the Message Map tab. The tab area that appears allows us to relate all of the controls that are clicked to message handlers in the new window class. You will need to create message handlers for all of the buttons. The method names to use for each of the appropriate IDs are:

IDC_BUY_CAVALRYS	=	OnBuyCavalrys
IDC_SELL_CAVALRYS	=	OnSellCavalrys

IDC_BUY_LEGIONS	=	OnBuyLegions
IDC_SELL_LEGIONS	=	OnSellLegions
IDC_BUY_ARCHERS	=	OnBuyArchers
IDC_SELL_ARCHERS	=	OnSellArchers
IDC_BUY_FLYERS	=	OnBuyFlyers
IDC_SELL_FLYERS	=	OnSellFlyers

Once the message maps are set up, click on the Member Variables tab. In this area, we are going to relate the four edit line controls that represent the number of playing pieces a player has purchased to a variable we can use in our code. The relationships to set up are:

IDC_BUY_CAVALRY_COUNT	=	m_buy_cavalry_count
IDC_BUY_LEGION_COUNT	=	m_buy_legion_count
IDC_BUY_ARCHER_COUNT	=	m_buy_archer_count
IDC_BUY_FLYER_COUNT	=	m_buy_flyer_count

That's it. Click on OK and then save and close AppStudio.

Filling in the Dialog Class

The Buy Equipment dialog box is ready to be displayed on the screen except it needs to have all of the message handlers filled. Each pair of message handlers works to buy and sell playing pieces. A playing piece is bought when the player clicks on the Buy button. The message handler must first check to see if the player has enough gold coins to purchase the playing piece. All of the playing pieces have a fixed value. These values are:

cavalry	=	20,000
legion	=	20,000
archer	=	10,000
flyer	=	40,000

If a playing piece can be purchased, a member value is incremented to indicate the purchase and the value of this variable is displayed in the edit line for the appropriate playing piece. Finally, the total number of gold coins for this player is decremented.

The sell message handler is much the same, except that the member variable is decremented, the new value is displayed, and the gold coins count is incremented to indicate that the playing piece was sold back.

The buy and sell relationship only works on the current selection of playing pieces. A player cannot bring up the Buy Equipment dialog box to sell pieces.

The first step is to add the member variables to the dialog box class. Open the file buy_equi.h and add the following variables to the class under the first PUBLIC heading

```
long total_gold_coins;
int cavalrys_bought,
    legions_bought,
    archers_bought,
    flyers_bought;
```

The first variable is the total number of gold coins the player has available for purchasing playing pieces. The next four variables are the member variables that will be used to keep track of all of the playing pieces purchased by a player.

Now we move to the buy_equi.cpp file. In the constructor for our dialog box, add the code

```
total_gold_coins = 0;
cavalrys_bought = 0;
legions_bought = 0;
archers_bought = 0;
flyers_bought = 0;
```

This code will initialize all of the member variables to zero. Next we move to the message handlers. We will show the message handlers for the cavalry playing pieces and the changes for the other three playing pieces.

Move to the method OnBuyCavalry and add the code:

```
 CEdit* temp;
 char buffer[12];
```

```
if (total_gold_coin >= 20000)
{
 cavalrys_bought++;
 total_gold_coins -= 20000;

 temp = (CEdit*)GetDlgItem(IDC_BUY_CAVALRYS_COUNT);
 sprintf(buffer, "%d", cavalrys_bought);
 temp->SetSel(0, -1, FALSE);
 temp->ReplaceSel(buffer);

 temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
 sprintf(buffer, "%ld", total_gold_coins);
 temp->SetSel(0, -1, FALSE);
 temp->ReplaceSel(buffer);

}
else
{
 MessageBox("You are out of money", "No More", MB_OK);
}
```

The code begins by checking to see if the user has enough gold coins to actually purchase the given playing piece. If the player does not have enough gold coins, a message box will be displayed. The message box tells the user that her or she is out of money and cannot purchase the playing piece.

If the player does have enough money, the code increments the member variable indicating the number of cavalry pieces purchased and subtracts the purchase price from the gold coins available. The next two sections of code should look familiar. The first section replaces the text in the edit control for the total pieces purchased with the current number of pieces purchased. The second section replaces the text for the total gold coins with the new value.

Now on to the sell method. Find the method labeled OnSellCavalry and add the code:

```
CEdit* temp;
char buffer[12];
if (cavalrys_bought == 0)
   MessageBox("You have not bought any cavalrys", "Cannot Sell", MB_OK);
else
{
 cavalrys_bought--;
 total_gold_coins += 20000;
```

```
    temp = (CEdit*)GetDlgItem(IDC_BUY_CAVALRYS_COUNT);
    sprintf(buffer, "%d", cavalrys_bought);
    temp->SetSel(0, -1, FALSE);
    temp->ReplaceSel(buffer);

    temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
    sprintf(buffer, "%ld", total_gold_coins);
    temp->SetSel(0, -1, FALSE);
    temp->ReplaceSel(buffer);
}
```

The code is basically the same as the buy method except it checks to make sure that the player has purchased at least one playing piece of this kind during the time this Buy Equipment dialog box has been displayed. If not, a message box is displayed telling the player the situation. Otherwise, the variables for this playing piece are decremented and the gold coins variable is incremented appropriately. The edit controls are then replaced with their appropriate values.

The message handlers for the other three playing pieces need to be filled with the same code, except that everywhere "cavalry" appears, it should be replaced with an "archer", "flyer", or "legion".

Interface to the View Object

The dialog box and its class are ready to be used. However, a large amount of additional code must be put into place for the view object to be of use. The first thing to change is the menu of the view object to include a place where players can click to bring up the Buy Equipment dialog box anytime they need it.

While in the Visual WorkBench, go to the AppStudio application. Click on the Menu resource type and then double-click on the menu that appears in the right part of the dialog box. You will get the dialog box shown in Figure 6.19.

Click on the Edit field and press the Insert key. A new space will open up. Double-click on the empty box and type in the name "Players" in the Caption edit line. Make sure that the radio box next to Popup is checked. A new empty box will appear under the Players menu item. Close the properties

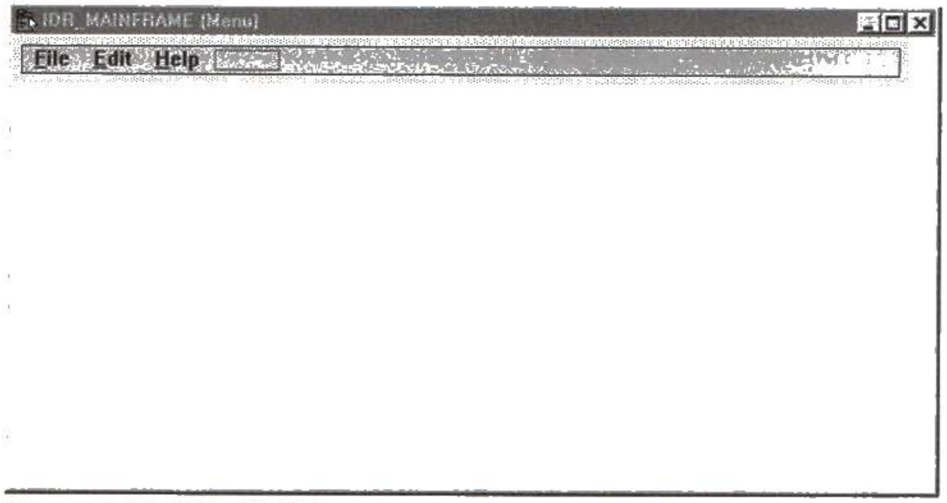


Figure 6.19 *IDR_MAINFRAME(Menu).*

box for the Players item. Double-click on the new empty box under the Players menu item. In the caption edit line, enter the name "Buy Equipment". Close the menu edit dialog box and then close AppStudio.

Now open ClassWizard. In the Class Name box, make sure that the class CNetwar2View is visible. In the Object ID list, you will find an ID called ID_PLAYERS_BUYEQUIPMENT. Click on this ID. Now double-click on the entry for COMMAND that will appear in the right list box. Click on the OK button of the dialog box that appears with the method name for this ID.

What we have just done is add a menu item to our view object. In ClassWizard, a message handler was created to handle a click on this menu item. This works just like a control in a dialog box. When a user clicks on the Buy Equipment menu item, it will send control to the OnPlayersBuyEquipment() method in the view class.

Close the ClassWizard and open the netwavw.cpp file. Find the method OnPlayersBuyEquipment() and add the following code.

```
buy_equipment pDlg;

pDlg.total_gold_coins = the_players[0]->gold_coins;
pDlg.DoModal();

the_players[0]->gold_coins = pDlg.total_gold_coins;
```

This code should look familiar. An object of the buy_equipment class is defined. The current player's total gold coins value is copied to the total gold coins variable in the Buy Equipment dialog box. Next, the dialog box is displayed on the screen with the DoModal() method. At this point, control is passed to the Buy Equipment dialog box until the OK or Cancel button is clicked. Once control is passed back to the view object, the current player's gold coins value is replaced with the number of gold pieces available after playing pieces have been purchased.

Save, build, and execute the current application. Once it is executed, click on the Players menu item followed by Buy Equipment. The Buy Equipment dialog box will appear on the screen. You can buy or sell any of the playing pieces. Click on OK when finished. The dialog box will disappear. The current code can be found in the directory /netwar2/chapt6/e.

Playing Pieces

Now it's time to actually do something with the playing pieces that are purchased by the player in the Buy Equipment dialog box. Obviously, the number of playing pieces that a user can purchase is only limited by the number of gold coins available to them. What are we going to do with all of the playing pieces? One of the obvious solutions is to put the playing pieces in the Armies list box on the operations window. However, if you remember from the code we have already written, each playing piece put in the Armies list box must have a CSprite class and associated variable defined. At this point in our code, we only have a single CSprite variable. In addition to the sprites that our player can purchase, we will have to keep track of all sprites for the other players, so we might as well define a variable that will keep track of all sprites.

First, we will add the variable that will handle all of the playing pieces purchased in the Buy Equipment dialog box as well as keep track of the playing pieces for other players. In the netwarvw.h file, add the following code in the first PRIVATE heading.

```
CSPRITE* playing_pieces[4][50];
int total_pieces[4];
```

This code will create a two-dimensional array. The first dimension is for the players, and the second dimension is for the playing pieces themselves. The second variable defined is used to keep track of the total number of playing pieces for each player.

In the constructor for this application, add the code:

```
total_pieces[0] = 0;
```

so that the starting count is zero for our player.

OnDraw() Method In order to use the new *playing_pieces* variable, we need to make a change to the OnDraw() function and all of the mouse button methods. In the OnDraw() function, change the code

```
if (cavalry_piece->active == TRUE)
   {
    if (cavalry_piece->show)
     cavalry_piece->Redraw(pDC);
    else
    {
     cavalry_piece->Start(pDC, 100, 100);
     cavalry_piece->show = TRUE;
    }
   }
```

to

```
for (j=0;j<1;j++)
{
      for(i=0;i<total_pieces[j];i++)
      {
       if (playing pieces[j][i]->active == TRUE)
       {
        if (playing_pieces[j][i]->show)
        {
         playing_pieces[j][i]->Redraw(pDC);
        }
        else
        {
         playing pieces[j][i]->Start(pDC, 50, 50+index0); //50 50
         playing pieces[j][i]->show = TRUE;
        }
       }
      }
}
```

The new code really isn't as complex as it looks. The outer loop runs through the players in the game. At this point, we are the only player. The code starts another loop that runs through all of the playing pieces for our player. The total number of pieces is kept in the appropriate array position of the variable *total_pieces*. For each playing piece of this player, the sprite is either displayed or not depending on the Active member variable for each piece.

After looking at this code, you get an idea about how the mouse click methods should appear. Instead of just looking at a single playing piece, we have to check for a click on each of the playing pieces for the player.

OnLButtonDown() Method Move to the OnLButtonDown()method and type the code

```
SpriteRect = new CRect(cavalry_piece->mX, cavalry_piece->mY,
      cavalry_piece->mX+cavalry_piece->mWidth, cavalry_piece->mY+cavalry_piece-
>mHeight);

  if ((cavalry_piece->war == FALSE) && (SpriteRect->PtInRect(point)))
  {
   SetCapture();

   GetClientRect(clientRect);
   MapWindowPoints(NULL, clientRect);
   topleft = SpriteRect->TopLeft();

   cavalry_piece->XOffset = point.x - topleft.x;
   cavalry_piece->YOffset = point.y - topleft.y;

   dragging = 1;
  }
```

with

```
int i;


  for (i=0;i<total_pieces[who_am_i];i++)
  {
   if (playing_pieces[who_am_i][i]->active == TRUE)
   {
     SpriteRect = new CRect(playing_pieces[who_am_i][i]->mX,
playing_pieces[who_am_i][i]->mY,
     playing_pieces[who_am_i][i[]->mX+playing_pieces[who_am_i][i]->mWidth,
playing_pieces[who_am_i][i]->mY+playing_pieces[who_am_i][i]->mHeight);

     if ((playing_pieces[who_am_i][i]->war == FALSE) && (SpriteRect-
>PtInRect(point)))
      {
        SetCapture();

        GetClientRect(clientRect);
        MapWindowPoints(NULL, clientRect);
        topleft = SpriteRect->TopLeft();

        playing_pieces[who_am_i][i]->XOffset = point.x - topleft.x;
        playing_pieces[who_am_i][i]->YOffset = point.y - topleft.y;

        dragging = i;
        return;
      }
    }
  }
```

The OnLButtonDown() method is responsible for checking if users have left clicked on a playing piece that they wish to move. Instead of just making a check of the cavalry piece, we have to check to see if any of the playing pieces has been clicked on. Outside of the outer loop and the cavalry piece variable being replaced with the *playing_pieces* variable, all of the code is the same.

OnRButtonDblClk() method Move to the OnRButtonDblClk() method and replace the code

```
SpriteRect = new CRect(cavalry_piece->mX, cavalry_piece->mY,
     cavalry_piece->mX+cavalry_piece->mWidth, cavalry_piece->mY+cavalry_piece-
>mHeight);

   if (SpriteRect->PtInRect(point))
   {
    // fill in player
    sprintf(string, "Owner: Player %d", 0);
    Dlg.m_stats_owner = string;

    // fill in type
    sprintf(string, "Type: %s", cavalry_piece->type);
    Dlg.m_stats_type = string;

    // fill in health
    sprintf(string, "Defense: %d units", cavalry_piece->defense);
    Dlg.m_stats_defense = string;

    // fill in strength
    sprintf(string, "Offense: %d units", cavalry_piece->offense);
    Dlg.m_stats_offense = string;
    int ret = Dlg.DoModal();
  }
```

with

```
int i;

   for (i=0;i<total_pieces[who_am_i];i++)
   {
    if (playing_pieces[who_am_i][i]->active == TRUE)
    {
      SpriteRect = new CRect(playing_pieces[who_am_i][i]->mX,
playing_pieces[who_am_i][i]->mY,
      playing_pieces[who_am_i][i[]->mX+playing_pieces[who_am_i][i]->mWidth,
playing_pieces[who_am_i][i]->mY+playing_pieces[who_am_i][i]->mHeight);

      if (SpriteRect->PtInRect(point))
      {
       // fill in player
       sprintf(string, "Owner: Player %d", playing_pieces[who_am_i][i]->player);
       Dlg.m_stats_owner = string;

       // fill in type
       sprintf(string, "Type: %s", playing_pieces[who_am_i][i]->type);
       Dlg.m_stats_type = string;

       // fill in health
       sprintf(string, "Defense: %d units", playing_pieces[who_am_i][i]->defense);
       Dlg.m_stats_defense = string;

       // fill in strength
       sprintf(string, "Offense: %d units", playing_pieces[who_am_i][i]->offense);
       Dlg.m_stats_offense = string;

       int ret = Dlg.DoModal();
       return;
      }
    }
   }
```

Again, the code is very similar to the original except we have an outer loop that checks for a click on each playing piece instead of just the cavalry piece. All of the information for the statistics dialog box is taken from the appropriate playing piece.

OnLButtonDblClk The same type of thing that we just did for the other mouse clicks has to be performed for the OnLButtonDblClk() method. Just add a loop around the other code and change all references to cavalry_piece to playing_pieces[who_am_i][i].

All of these changes in our code allow us to click and move any of our playing pieces that were purchased from the Buy Equipment dialog box. The last thing we have to do is create all of the playing pieces that the player purchases and place them in the Armies list box of the operations window.

Interface to the Operations Window

In order to get the playing pieces that the player has purchased to the Armies list box on the operations window, two steps are required. The first step is to update the code for storing the armies in the list box, and the second is to create and send the playing pieces to the list box.

First we will update the code in the Armies list box method. Open the file operatio.cpp and replace all of the code in the OnAddArmies() method with the code

```
int i;
CListBox* box;

   box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
   box->ResetContent();
   for(i=0;i<total_armies;i++)
   {
    if (list_box[i] == -1)
      box->AddString("Army Deployed");
    else
      box->AddString(entries[i]);
   }

   return 1;
```

When the operations window is sent a message to add an army to the list box, it will just run through an array that contains the name of the playing piece to display. It will determine whether the name of the playing piece is displayed or the words "Army Deployed" based on the entry in the *list_box* array.

Now open the operatio.h file and find the place where we define the variable *list_box*. Increase the array entry to 50. Find the entry for the variable *entries*. Change it to read

```
char entries[50][20];
```

In the first change, we increased the hold capacity of the *list_box* array to 50 so that this player can handle up to 50 playing pieces. In the second change, we created a two-dimensional array of 50 spaces for 20 character strings. This is where the name of the playing piece will be displayed.

Therefore, to add an army to the list box, the code must:

1. place the name of the playing piece in the array position,

   ```
   entries[total_armies]
   ```

2. place the value 0 in the array position,

   ```
   list_box[total_armies]
   ```

3. increase the total_armies variable by one,
4. send a WM_OPERATIONS_ADD_ARMIES message to the operations window.

Now it's time for the second change to the code, which is to process the playing pieces that the player purchased.

The first thing we need to do is change the way our playing pieces are brought into the system. Instead of assigning a single variable to each bitmap: normal, mask, and highlighted, a bitmap array will be created to hold all of the bitmaps.

In the netwavw.h file, add the following code in the first PRIVATE area.

```
Cbitmap* bitmaps[48];
```

This code creates an array called *bitmaps* that can hold up to 48 bitmap objects. In order to keep track of which array positions hold what bitmaps, several constants will be used. At the top of this file, add the code

```
#define CAVALRY_BITMAP 0
#define CAVALRY_BITMAP_MASK 1
#define CAVALRY_BITMAP_HIGHLIGHT 2
```

These defined constants can now be used to reference the specific bitmaps in each array position. Later in this chapter we will see that additional bitmaps will be needed for our remaining playing pieces.

With the bitmap array, the code that reads in the bitmaps needs to be changed. Open the netwavw.cpp file and look in the constructor code. Replace the code that reads in the bitmaps with

```
//Load bitmaps
   bitmaps[CAVALRY_BITMAP] = new CBitmap;
   if (bitmaps[CAVALRY_BITMAP]->LoadBitmap(IDB_CAVALRY_BITMAP) == FALSE)
    MessageBox("Unable to load cavalry bitmap", "error", MB_OK);

   bitmaps[CAVALRY_BITMAP_MASK] = new CBitmap;
   if (bitmaps[CAVALRY_BITMAP_MASK]->LoadBitmap(IDB_CAVALRY_BITMAP_MASK) == FALSE)
    MessageBox("Unable to load cavalry mask bitmap", "error", MB_OK);

   bitmaps[CAVALRY_BITMAP_HIGHLIGHT] = new CBitmap;
   if (bitmaps[CAVALRY_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_CAVALRY_BITMAP_HL) == FALSE)
    MessageBox("Unable to load cavalry highlight bitmap", "error", MB_OK);
```

This new code will load the bitmaps for the cavalry playing piece into the 0, 1, and 2 array positions. Just after this code, you will find additional code that creates a playing pieces. Delete this code from the constructor. Further down in the constructor, you will find the code

```
strcpy(ops_dlg->entries, "cavalry");
ops_dlg->SendMessage(WM_OPERATIONS_ADD_ARMIES, 0, 0L);
```

Remove this code as well.

Reading the Playing Pieces Purchased In the netwavw.cpp file, find the OnPlayersBuyEquipment() and replace the code in it with the code

```
buy_equipment pDlg;


    pDlg.total_gold_coins = players[0]->gold_coins;

    pDlg.DoModal();

    players[0]->gold_coins = pDlg.total_gold_coins;
    m_pDlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);

    if (pDlg.cavalrys_bought > 0)
    {
        for(i=0;i<pDlg.cavalrys_bought;i++)
        {
         playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
         if (playing_pieces[who_am_i][total_pieces[who_am_i]]->Initialize(GetDC(),
bitmaps[CAVALRY_BITMAP_MASK],    bitmaps[CAVALRY_BITMAP],
          bitmaps[CAVALRY_BITMAP_HIGHLIGHT],
             FALSE, 4, 3, who_am_i, "Cavalry", 4) == FALSE)
           MessageBox("Unable to initialize dragon sprite", "Error", MB_OK);

          strcpy(ops_dlg->entries[ops_dlg->total_armies], "Cavalry");
          ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
          ops_dlg->total_armies++;

          total_pieces[who_am_i]++;
      }
  }

  ops_dlg->SendMessage(WM_OPERATIONS_ADD_ARMIES, 0, 0L);
```

The object of this code is to read the number of cavalry playing pieces the player purchased, create the playing pieces, and display their names in the add_armies list box in the operations window. Since the player is able to purchase more than just one cavalry playing piece at a time, the code has to keep track of each individual playing piece.

To determine if the player has purchased cavalry playing pieces, a check is made of the variable *pDlg.cavalrys_bought*. Recall that this variable is a member of the buy_equipment window class. If its value is greater than one, we know that cavalry pieces were purchased. The code immediately enters a loop based on the number of pieces purchased.

Inside the loop, the code creates a new sprite object in the *playing_pieces* array. Notice that the sprite objects are places in the *who_am_i* position of the array. Next, the current playing piece is initialized and created. All of the playing pieces are set up as not active. The next three lines of code set up the Armies list box on the operations window. The name of the playing piece is copied into the next available list box slot. The number of this playing piece is stored in the *list_box* variable, and the total number of playing pieces in the Armies list box is incremented. Finally, the code increments the total number of playing pieces for this player.

The code will continue to do these steps for the total number of cavalry playing pieces that the player purchased. After all of the playing pieces have been created, initialized, and added to the Armies list box variables, a message is sent to the operations window indicating that the list box should be displayed in the operations window.

Selecting a Playing Piece The game allows us to purchase playing pieces and puts them in the Armies list box of the operations window. The only thing we have to concern ourselves with right now is what happens when an army is double-clicked on in the list box. When this happens, the operations window code will change the name of the army to "Army Deployed" and set the variable *selected_army* to the playing pieces index value for the *playing_pieces* array. We can use this value to change the active status from FALSE to TRUE. In the method *long CNetwar2View::OnArmySelected(UINT wParam, LONG lParam)* change the code to read

```
{
 playing_pieces[who_am_i][ops_dlg->selected_army]->active = TRUE;
 Invalidate(TRUE);

 return 1;
}
```

We have removed the reference to the *cavalry_piece* variable and put in a reference to our *playing_pieces* array. Since we are dealing with our own playing pieces, we use the *who_am_i* variable to select the first list of playing pieces. The *selected_army* variable is used from the operations window to select the correct playing piece to change from FALSE to TRUE.

By putting an Invalidate(TRUE) statement here in this method, the screen will be redrawn and the playing piece just selected will be placed on the playing field.

Code The code and executables for the game to this point can be found in the /chapt6/f directory. Execute the program and select the Buy Equipment menu option. Purchase a couple of cavalry playing pieces and click OK on the window. The names of the playing pieces that you purchased will appear in the Armies list box in the operations window. Double-click on one of the armies and move it around the playing field. Now go back and double-click on a second army. Notice that you can move either of the armies, select them for war, or display their statistics. All of the playing pieces are independent.

Remaining Playing Pieces

Everything we have done with the playing pieces works. Now we need to expand the system code to accommodate the other three playing pieces. To do this, we have to create and draw the playing pieces using AppStudio, assign constants to each bitmap, and read them into the game.

Creating the Remaining Bitmaps We have only created bitmaps for our cavalry playing piece. We need to have bitmaps for the flyer, legion, and archer pieces are well. Using AppStudio, create normal, mask, and highlighted bitmaps for each of the remaining pieces. The bitmaps should be given the IDs

flyer normal	IDB_FLYER_BITMAP
flyer mask	IDB_FLYER_BITMAP_MASK
flyer highlighted	IDB_FLYER_BITMAP_HL
archer normal	IDB_ARCHER_BITMAP
archer mask	IDB_ARCHER_BITMAP_MASK
archer highlighted	IDB_ARCHER_BITMAP_HL
legion normal	IDB_LEGION_BITMAP
legion mask	IDB_LEGION_BITMAP_MASK
legion highlighted	IDB_LEGION_BITMAP_HL

Once all of the bitmaps are finished, constants are created for them.

Creating Constants Open the netwarvw.h file and add the following constant names to the top of the file along with the cavalry bitmap constants:

```
#define FLYER_BITMAP                  3
#define FLYER_BITMAP_MASK             4
#define FLYER_BITMAP_HIGHLIGHT        5

#define ARCHER_BITMAP                 6
#define ARCHER_BITMAP_MASK            7
#define ARCHER_BITMAP_HIGHLIGHT       8

#define LEGION_BITMAP                 9
#define LEGION_BITMAP_MASK           10
#define LEGION_BITMAP_HIGHLIGHT      11
```

These constants will be used in the same capacity as the cavalry constants.

Reading the Bitmaps into the Game With the bitmaps designed and constants available, it's time to read all of the bitmaps into the game. Add the following code to the view's constructor:

```
bitmaps[FLYER_BITMAP] = new CBitmap;
if (bitmaps[FLYER_BITMAP]->LoadBitmap(IDB_FLYER_BITMAP) == FALSE)
   MessageBox(“Unable to load flyer bitmap”, “error”, MB_OK);

bitmaps[FLYER_BITMAP_MASK] = new CBitmap;
if (bitmaps[FLYER_BITMAP_MASK]->LoadBitmap(IDB_FLYER_BITMAP_MASK) == FALSE)
   MessageBox(“Unable to load flyer mask bitmap”, “error”, MB_OK);

bitmaps[FLYER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[FLYER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_FLYER_BITMAP_HL) == FALSE)
   MessageBox(“Unable to load flyer highlight bitmap”, “error”, MB_OK);

bitmaps[ARCHER_BITMAP] = new CBitmap;
if (bitmaps[ARCHER_BITMAP]->LoadBitmap(IDB_ARCHER_BITMAP) == FALSE)
   MessageBox(“Unable to load archer bitmap”, “error”, MB_OK);

bitmaps[ARCHER_BITMAP_MASK] = new CBitmap;
if (bitmaps[ARCHER_BITMAP_MASK]->LoadBitmap(IDB_ACHER_BITMAP_MASK) == FALSE)
   MessageBox(“Unable to load archer mask bitmap”, “error”, MB_OK);

bitmaps[ARCHER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[ARCHER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_ACHER_BITMAP_HL) == FALSE)
   MessageBox(“Unable to load archer highlight bitmap”, “error”, MB_OK);

bitmaps[LEGION_BITMAP] = new CBitmap;
if (bitmaps[LEGION_BITMAP]->LoadBitmap(IDB_LEGION_BITMAP) == FALSE)
   MessageBox(“Unable to load legion bitmap”, “error”, MB_OK);

bitmaps[LEGION_BITMAP_MASK] = new CBitmap;
if (bitmaps[LEGION_BITMAP_MASK]->LoadBitmap(IDB_LEGION_BITMAP_MASK) == FALSE)
   MessageBox(“Unable to load legion mask bitmap”, “error”, MB_OK);

bitmaps[LEGION_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[LEGION_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_LEGION_BITMAP_HL) == FALSE)
   MessageBox(“Unable to load legion highlight bitmap”, “error”, MB_OK);
```

Just for a simple game with one player and four playing pieces, we have to read in a total of 12 bitmaps. Once we add the other three players, this number will jump to 48 bitmaps.

Final Buy Equipment Code

Once the new playing piece bitmaps have been added to the system, constants defined, and the bitmaps read into the game, we can add final code to the buy_equipment window method to create these new playing pieces when a player purchased them. The following code should be added just after the cavalry buy code and before the SendMessage to the operations window:

```
if (pDlg.flyers_bought > 0)
   {
      for(i=0;i<pDlg.flyers_bought;i++)
      {
      playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
      if (playing_pieces[who_am_i][total_pieces[who_am_i]]-
>Initialize(GetDC(), bitmaps[FLYER_BITMAP_MASK],bitmaps[FLYER_BITMAP],
                                     bitmaps[FLYER_BITMAP_HIGHLIGHT],
                              FALSE, 4, 3, who_am_i, "Flyer", 4) == FALSE)
       MessageBox("Unable to initialize flyer sprite", "Error", MB_OK);

      strcpy(ops_dlg->entries[ops_dlg->total_armies], "Flyer");
      ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
      ops_dlg->total_armies++;

      total_pieces[who_am_i]++;
     }
   }
if (pDlg.archers_bought > 0)
   {
      for(i=0;i<pDlg.archers_bought;i++)
      {
      playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
      if (playing_pieces[who_am_i][total_pieces[who_am_i]]-
>Initialize(GetDC(), bitmaps[ARCHER_BITMAP_MASK], bitmaps[ARCHER_BITMAP],
                                     bitmaps[ARCHER_BITMAP_HIGHLIGHT],
                              FALSE, 4, 3, who_am_i, "Archer", 4) == FALSE)
      MessageBox("Unable to initialize archer sprite", "Error", MB_OK);

      strcpy(ops_dlg->entries[ops_dlg->total_armies], "Archery");
      ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
      ops_dlg->total_armies++;

      total_pieces[who_am_i]++;
    }
   }
if (pDlg.legions_bought > 0)
   {
      for(i=0;i<pDlg.legions_bought;i++)
      {
      playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
      if (playing_pieces[who_am_i][total_pieces[who_am_i]]-
>Initialize(GetDC(), bitmaps[LEGION_BITMAP_MASK], bitmaps[LEGION_BITMAP],
                                        bitmaps[LEGION_BITMAP_HIGHLIGHT],
                                  FALSE, 4, 3, who_am_i, "Legion", 4) == FALSE)
      MessageBox("Unable to initialize legion sprite", "Error", MB_OK);

      strcpy(ops_dlg->entries[ops_dlg->total_armies], "Legiony");
      ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
      ops_dlg->total_armies++;

      total_pieces[who_am_i]++;
   }
  }
```

The code is exactly the same as the buy code for the cavalry playing piece, except we have to make sure that we are looking at the correct bought variable for each playing piece.

Once you have added all of this code to the game, compile and execute it. You will be able to purchase any of the four playing pieces and put them up on the playing field. The final code for this chapter can be found in the directory /chapt6/g.

Summary

Our game is getting more and more complete. This chapter has added the operations window and the Buy Equipment dialog box. These two additions required the updating of most code for allowing access to multiple playing pieces instead of just one. A player can purchase many different playing pieces, place them all on the playing field at the same time, and manipulate them appropriately.

Chapter 7

Game Details

Before we get into the communication and multiplayer aspects of our game, several details need to be cleaned up. This chapter will deal with those details. The topics include a war engine that will determine the winner of a war, adding sound effects, limiting playing piece movement, and allowing a new game to be played.

War Engine

The most important part of our game is the war mechanism between playing pieces. The system needs to be balanced and fair—it should take into account the relative strengths of the units while providing some element of chance. A player with a very strong unit should feel confident that he or she can defeat a weak unit *most of the time*. An element of chance in the war mechanism introduces much more suspense and realism into the game.

Recall that each playing piece has two point variables called defense and offense. The offense points are the strength of the playing piece when it attacks another, and the defense points are the strength when the playing piece is defending an attack. The points given to each playing piece must be appropriate for the type of battle unit it represents.

The cavalry playing piece can be used as a reference point for the other playing pieces. The cavalry playing piece will have points of 4 for offense and 3 for defense. Since the cavalry unit is riding horses, it has speed and power on offense, but for the same reason has slightly less defense because it can't easily hide from an attacker.

The legion is a larger force than the cavalry and does not ride horses. Their points will be 8 and 6. The archers will be the attacking unit playing piece, with points of 4 and 1. They are light on their feet and able to attack fearlessly, but don't carry much armor. The flyers are a level group with points of 3 and 3.

With each playing piece having specific offense and defense points, it seems natural to make our war mechanism use a rule such as:

```
IF OFFENSE of attacker > DEFENSE of attacked, THEN attacker WINS!!!
```

This would sure be a fun game to play!¿ In a matter of minutes, the playing field would be cleared of playing pieces and a winner announced. The attacker would almost always win the battle. Would an archer really attack a legion¿

What is needed is a more intelligent war mechanism. A random value could be put into the rule such as:

```
IF OFFENSE of attacker > DEFENSE of attacked AND random value > 4, THEN attacker
WINS!!
```

This type of rule would add a surprise factor to the outcome of a war. One of the problems with the rule, though, is the fact that a legion should destroy an archer unit in a single battle. The legion should not have to continue to engage the archer unit a number of times just to get a lucky roll of the random number generator.

An additional problem with each of these rules is simplicity. There is no provision in the rules for just hurting the playing piece and not destroying it. It might be an added touch if a unit could be half damaged or not even damaged at all.

War Matrix

The solution that will be used in this game is called a "war matrix". The purpose of the war matrix is to determine the outcome of a war based on the offense points of the attacker, the defense points of the attacked, and a value randomly generated by the computer. What sets the war matrix apart from the simple rules given earlier is that the designer of the matrix is able to weight the outcome of a war regardless of the random value.

The war matrix itself consists of a three-dimensional matrix that measures 12 by 12 by 6. The outer dimensions of 12 by 12 relate to the maximum offense and defense points the game allows. You will recall that the highest number of points assigned to any of our playing pieces was 8, so why go up to 12? The reason is a playing piece will be able to earn additional points based upon the number of playing pieces it has been able to destroy. In addition, we will be adding points for those playing pieces that are defending their home castles.

Matrix

As mentioned, the matrix is 12 by 12 by 6. The definition of the array looks like

```
int war_matrix[12][12][6];
```

The first dimension of the array is accessed using the offense value of the playing piece starting the war. The second dimension of the array is accessed using the defense value of the playing piece being engaged. These two values put us at a specific location in the matrix. This location is a third dimension, and has six locations in it.This is where a random number comes into play. Once the first two dimensions are determined, a random number picks the third. The value at the third dimension will be the outcome of the war.

In the third dimension is a list of six values. Each value corresponds to a possible outcome to the current battle. We can set up a list of constants like

```
#define NO_DAMAGE 0
#define HALF_DAMAGE 1
#define BOTH_DAMAGE 2
#define ATTACKER_DAMAGE 3
#define DESTROYED 4
```

One of these constants would be placed in each of the six third-dimension positions. Our random number will determine which of the outcomes will be used.

For instance, let's say that a war is taking place between a flyer and a legion. The flyer is the attacker. The matrix position for the war would be:

```
war_matrix[3][6][random]
```

because the flyer's offense value is 3 and the legion's defense value is 6. Now the random number has to be determined.

The six locations at [3][6] might be something like:

```
war_matrix[3][6][0] = 0
war_matrix[3][6][1] = 1
war_matrix[3][6][2] = 0
war_matrix[3][6][3] = 3
war_matrix[3][6][4] = 2
war_matrix[3][6][5] = 0
```

As you can see from the list of numbers that have been put into the six locations, there is a better chance that no damage will occur to the attacked. There is no chance that the attacker will be damaged. By using this type of war engine, the designer can make the outcomes of a war relate to the strength of each playing piece.

In our example, we are just waiting for a random number to be picked. The random number might turn out to be 4. In this case, we would move to the location [3][6][4] and get the value in the matrix. The value is 2, which means that both of the playing pieces must accept damage.

Adding Matrix Code

Now that you know how the matrix works, its time to add the necessary code to our game. The first step is to create a new object class for the war matrix. In the Visual Workbench, open a new file and place the code:

```
#define NO_DAMAGE 0
#define HALF_DAMAGE 1
#define BOTH_DAMAGE 2
#define ATTACKER_DAMAGE 3
#define DESTROYED 4
class CBattle
{
  public:
      get_battle_result(int offense, int defense);
   private:
    int battle_matrix[12][12][6];

   public:
    CBattle();
    ~CBattle();
};
```

Save this file as warmatrx.cpp.

The code at the top of the file is just the war outcome constants that were mentioned above. You can have basically any number of possible outcomes for a war. After the outcome constants, we have the class definition for the war matrix.

There are three PUBLIC members for the class: a constructor, a destructor, and get_battle_result. The get_battle_result() method will return the war outcome value based on the given offense and defense values. The random number part of the outcome is handled by this method.

Next, we have the PRIVATE part of the class. This is the matrix itself. It is defined as a 12 by 12 by 6 matrix that can hold integers.

Close this file and open a new file. Place the following code in the file.

```
#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
#include "netwavw.h"
#include <time.h>
CBattle::CBattle()
{
  int i, j, k;

  srand((unsigned) time(NULL));
  //zero out battle matrix
  for(i=0;i<12;i++)
   for(j=0;j<12;j++)
    for(k=0;k<6;k++)
     battle_matrix[i][j][k] = NO_DAMAGE;

  //0 to 0
  battle_matrix[0][0][0] = NO_DAMAGE;
  battle_matrix[0][0][1] = BOTH_DAMAGE;
  battle_matrix[0][0][2] = NO_DAMAGE;
  battle_matrix[0][0][3] = BOTH_DAMAGE;
```

```
battle_matrix[0][0][4] = NO_DAMAGE;
battle_matrix[0][0][5] = NO_DAMAGE;

//0 to 1
battle_matrix[0][1][0] = NO_DAMAGE;
battle_matrix[0][1][1] = BOTH_DAMAGE;
battle_matrix[0][1][2] = ATTACKER_DAMAGE;
battle_matrix[0][1][3] = NO_DAMAGE;
battle_matrix[0][1][4] = NO_DAMAGE;
battle_matrix[0][1][5] = ATTACKER_DAMAGE;

//0 to 2
battle_matrix[0][2][0] = BOTH_DAMAGE;
battle_matrix[0][2][1] = NO_DAMAGE;
battle_matrix[0][2][2] = ATTACKER_DAMAGE;
battle_matrix[0][2][3] = ATTACKER_DAMAGE;
battle_matrix[0][2][4] = NO_DAMAGE;
battle_matrix[0][2][5] = ATTACKER_DAMAGE;

//0 to 3
battle_matrix[0][3][0] = NO_DAMAGE;
battle_matrix[0][3][1] = NO_DAMAGE;
battle_matrix[0][3][2] = ATTACKER_DAMAGE;
battle_matrix[0][3][3] = NO_DAMAGE;
battle_matrix[0][3][4] = NO_DAMAGE;
battle_matrix[0][3][5] = ATTACKER_DAMAGE;

//0 to 4
battle_matrix[0][4][0] = NO_DAMAGE;
battle_matrix[0][4][1] = NO_DAMAGE;
battle_matrix[0][4][2] = NO_DAMAGE;
battle_matrix[0][4][3] = NO_DAMAGE;
battle_matrix[0][4][4] = NO_DAMAGE;
battle_matrix[0][4][5] = ATTACKER_DAMAGE;

for(i=5;i<12;i++)
  for(j=0;j<6;j++)
    battle_matrix[0][i][j] = NO_DAMAGE;

//1 to 0
battle_matrix[1][0][0] = HALF_DAMAGE;
battle_matrix[1][0][1] = BOTH_DAMAGE;
battle_matrix[1][0][2] = ATTACKER_DAMAGE;
battle_matrix[1][0][3] = HALF_DAMAGE;
battle_matrix[1][0][4] = NO_DAMAGE;
battle_matrix[1][0][5] = HALF_DAMAGE;

//1 to 1
//below

//1 to 2
```

```
    battle_matrix[1][2][0] = HALF_DAMAGE;
    battle_matrix[1][2][1] = BOTH_DAMAGE;
    battle_matrix[1][2][2] = ATTACKER_DAMAGE;
    battle_matrix[1][2][3] = HALF_DAMAGE;
    battle_matrix[1][2][4] = NO_DAMAGE;
    battle_matrix[1][2][5] = HALF_DAMAGE;


    for(i=0;i<12;i++)
    {
     battle_matrix[i][i][0] = HALF_DAMAGE;
     battle_matrix[i][i][1] = BOTH_DAMAGE;
     battle_matrix[i][i][2] = HALF_DAMAGE;
     battle_matrix[i][i][3] = NO_DAMAGE;
     battle_matrix[i][i][4] = NO_DAMAGE;
     battle_matrix[i][i][5] = NO_DAMAGE;
    }
}


CBattle::get_battle_result(int offense, int defense)
{
   srand((unsigned) time(NULL));
   return battle_matrix[offense][defense][rand() % 6];
}
```

The file should be called warmatrx.cpp and saved. There are only two methods in this code: the constructor and get_battle_result(). The constructor for the war matrix is responsible for setting up the matrix. It begins by placing the NO_DAMAGE value in all of the array positions. Next, each of the outcomes for all offense/defense value combinations is assigned a specific outcome. This code only shows part of the actual initialization. You will want to fill in all of the matrix locations with outcome values.

The get_battle_result() method is at the bottom of the code. The function begins by seeding the C random number generator with the current time. This will cause a truly random number to be generated. Next, the code returns the value located at the appropriate offense/defense/random location in the matrix.

Game Integration

This is the complete war matrix. Now we need to look at interfacing the matrix into our game code. The first thing is to let the project file know about the new source code. Click on Project and Edit. Add the warmatrx.cpp file and click on Close.

The war matrix will need to be created by the view object and stored somewhere. Open the netwavw.h file and add the following code to the top of the file:

```
#include "warmatrx.h"
```

Now move down to the view class definition and add the following code to the first public area:

```
CBattle* war_matrix;
```

This code will create an object pointer that we can use to create and initialize a war matrix object. Close this file and open the netwavw.cpp file. Move to the end of the view constructor and add the code:

```
// Create war matrix
war_matrix = new CBattle;
```

This code will create a new war matrix object and automatically initialize it for us. We can now use the matrix to find the outcome of a war. We know that a war takes place when a player highlights two playing pieces and clicks on the war button. Since we don't have two players in our game yet, we will have to postpone the discussion of using the war matrix until chapter 10.

Castle Points

It was mentioned above that there will be cases when the offense/defense value will be changed during a war. One of the changes occurs when the playing piece being attacked is located in a player's castle. Each player has a castle on the screen from where the player's pieces appear when placed on the playing field. Our war mechanism code will have to check to make sure that the attacked piece is not in its castle. If the playing piece is in its castle, the piece's defense points will be doubled.

Kill Points

A playing piece can also have its offense points increased through war. Each time a playing piece is used in a battle and destroys the opponent, an internal counter will be increased. When this counter is greater than four, one point will be added to the playing piece's offense value. To keep track of the number of kills, we will need to add a member variable to our Cplayer class. Open the cplayer.h file and add the statement:

```
int kills;
```

to the first PUBLIC area.

Open the file cplayer.cpp and add the statement:

```
kills = 0;
```

to the constructor. In Chapter 12, we will use this variable to keep track of our kills.

You will find the source code and executables with the war engine added in the netwar2/chapt7/a directory on the enclosed CD-ROM.

Sound Effects

Almost all games can benefit from the addition of sound. The game we are designing can also benefit from sound. The Windows environment makes adding sound fairly easy. Using function calls defined in the SDK for Windows, we can play sound through a number of different output devices.

When a sound device is installed in Windows, it can be selected as the primary sound output device. This means that both a speaker driver and a SoundBlaster driver can reside on the same PC with one of them selected as the default sound device. The SDK function sndPlaySound() can be used to play a sound through the default sound device.

Adding sound to the game involves several steps:

1. Find a sound.

2. Add the sound to the resource file.
3. Find the sound on the disk.
4. Load the sound into memory.
5. Play the sound.

Finding a Sound

When we speak of finding a sound, we mean actually finding or recording a sound that will be usable in our program. The Microsoft Windows system prefers the use of .wav files for sound. Therefore, you will need to find sounds that are recorded in this format.

There are several ways to find sounds. The first is to record them yourself using a sound card. Another way to find sound files is to look on the Internet. It should be noted that many of the sounds available on that forum are in violation of somebody's copyright, so you will want to be careful using these sounds. Still another way to find sounds is to purchase a CD-ROM that contains many different sounds.

The sounds that you find will need to relate to some action in the game that you feel will be better if sound effects are added to it. Examples are when an army is placed on the playing field and when an army is destroyed.

Add Sound to a Resource File

You should now have the sound files that you want to use in your game. For the examples that follow, we will be using a sound file called "test.wav". This sound file will be used when an army is added to our playing field.

The first step in adding this sound to our game is to put it in the game's resource file. Unfortunately, the current Visual C++ system, specifically App-Studio, doesn't know what to do with this sound file, so we have to manually add the sound to the resource file. To make the process easier, we will create a separate resource file specifically for sounds. The new resource file will be linked or included in the game's main resource file so that it is automatically compiled.

In the Visual WorkBench, open a new file and add the following code to it.

```
#ifdef APSTUDIO_INVOKED
#error : Do not edit this file with App Studio
#endif
ADD_ARMY_SOUND        sound    test.wav
```

Close the file and save it under the name sounds.rc. This is a sound resource file that we can use to add any number of sounds to the game. The first three lines tell the AppStudio application that this resource file cannot be edited. When you try to load this file into AppStudio, you will get the error message "Do not edit this file with App Studio".

After the first three lines, the sounds that will be used in the game are listed. The first entry is the resource name of the sound. This is the name that you will use in the game's code to relate to this sound. The second entry is the word "sound". This tells the resource compiler that this is a sound resource. The last entry is the filename of the sound on the disk. The system will try to load the sound based on this entry. You will need to specify paths to the sound if necessary.

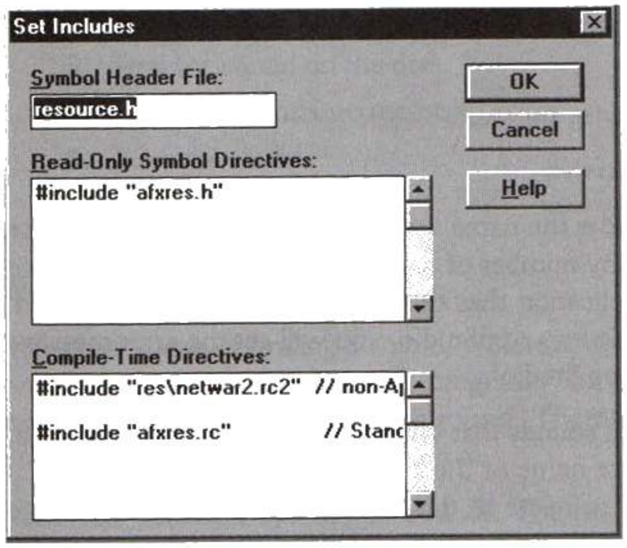
Now we need to tell the main resource file about the sound resource file. Start AppStudio and load the game's main resource file. Click on File and then Set Includes. You should see the options dialog box as shown in Figure 7.1.

In the list box labeled Compile-Time Directives, you will see a space between the current two entries. Add the following line of code in the empty space.

```
#include "sounds.rc"    //non-AppStudio edited resources
```

Click on OK, then read the warning and click on its OK. What we have done is add an *include* statement to the game's main resource file that will pull our sound resource file into itself during compile time. This will cause all of the resources to be automatically compiled.

Before we leave AppStudio, we have to do one other thing. The name that we gave our sound file has to relate to some number. To do this, click on the Edit menu option in AppStudio and then on Symbols. The symbols are the names given to all of the resources in our game. Click on New. Enter the name ADD_ARMY_SOUND and click on OK. Our sound file and its name will be given a number that AppStudio uses to keep track of the resources.



Symbol Header File: resource.h

Read-Only Symbol Directives:

```
#include "afxres.h"
```

Compile-Time Directives:

```
#include "res\netwar2.rc2"  // non-A
#include "afxres.rc"         // Stanc
```

Figure 7.1 *Set Includes dialog box.*

When you build the game again, you must make sure that the file test.wav is in the same directory as the sounds.rc file.

Find and Load Sound

Adding the sound to the resource file will cause the sound file to be included in the final .exe executable file and provide a name, ADD_ARMY_SOUND, which we can use to reference our sound in the game's main code. Now we have to find and load the sound into memory so it can be played.

The first step is to define two variables for our sound. Open the netwavw.h file and add the following code to the first PRIVATE area.

```
HRSRC add_army_sound;
HGLOBAL add_army_sound_load;
```

The first line of code defines a variable called *add_army_sound* that will be used as a resource handle. The second line of code defines a variable called *add_army_sound_load* that will be used as a handle to be global memory object. Remember we said that we would load the sound into memory so that we can play it.

Each sound that is used in the game will have to have two variables like these. The variables will be used in the actual loading of the sound. Open the netwavw.cpp file and add the following code in the constructor for our view object:

```
//load war sound
   add_army_sound = ::FindResource(AfxGetResourceHandle(),
                              (LPCSTR)MAKEINTRESOURCE(ADD_ARMY_SOUND), "sound");
   add_army_sound_load = ::LoadResource(AfxGetResourceHandle(), add_army_sound);
```

The first line of code uses the method FindResource() to locate the sound resource in the executable file. Of particular importance are the second and third parameters in this method.The second parameter is the name of the sound we placed in the sound resource file. This is the first entry for each sound. You will want to make sure that the name of the sound is spelled exactly as it is in the sound resource file. The third parameter is the type of resource the method is trying to find. By putting in the name "sound", we make sure that the method will only search the resources in our sound resource file, since those are the only resources designated as sound.

Once the sound is found by the FindResource() method, a pointer or handle is made to the location of the sound and assigned to the *add_army_sound* variable. The second line of code will use this handle to load the sound into memory. Notice that the *add_army_sound* variable is passed as the second parameter in the LoadResource method.

After these two methods have executed, the variable *add_army_sound_load* contains a handle to the memory holding our sound file. Now all we need to do is play the sound.

Play Sound

Playing the sounds that we have loaded into memory is a simple process. Find the location in the game code where you would like the sound to be heard and place the code:

```
sndPlaySound((LPCSTR)::LockResource(add_army_sound_load),
                                    SND_MEMORY|SND_ASYNC|SND_NODEFAULT);
UnlockResource(add_army_sound_load);
```

The sound is played using the sndPlaySound() function. Before the sound can actually be played, though, the sound has to be locked into memory. This is performed with the LockResource() method. The first parameter in this function is the handle to the memory of the sound that we want to play. The second parameter is a flag. In our case, we are telling the function that our sound is already in memory and that the sound should be locked in such a way that it can be played synchronously or while other system activities are happening.

After the sound is locked in memory, it is played. Once played, the sound is unlocked with the UnlockResource() method. To hear the sound in action, add the preceding two lines of code in the OnArmySelected() method in the netwavw.cpp file. Every time the player adds an army to the playing field, our sound will be played.

The last thing we need to do to play our sound is add an *include* statement to the top of the netwavw.cpp file. The statement to add is:

```
#include <mmsystem.h>
```

This causes the code necessary for the sndPlaySound() method to be available to the compiler. We also need to have the code available to the linker. Click on the Options menu item in the Visual WorkBench, then click on Project. You should see the dialog box illustrated in Figure 7.2. Click on the button for Linker to get the dialog box shown in Figure 7.3.

In the lower-left corner of the dialog box is a list of category options. Click on Windows Libraries to bring up the dialog box in Figure 7.4. In the list box on the right hand side of the dialog box, you will see an entry for MMSYSTEM. Click on this entry to highlight it. This indicates to the linker that you wish the MultiMedia system to be loaded with your application. This system is necessary for the sndPlaySound() function to operate correctly. Now click on OK twice.

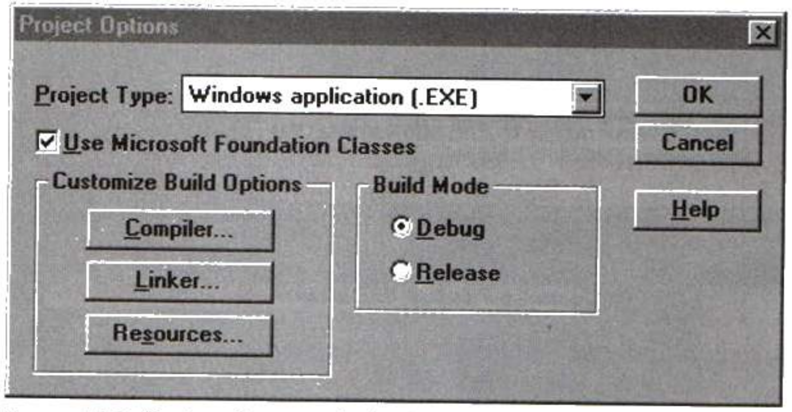


Figure 7.2 *Project Options dialog box.*

Using the Speaker Beep

In addition to the sound file, we have one other sound that can be used in our program. This sound is a beep from the computer's speaker. The beep can be created with the statement:

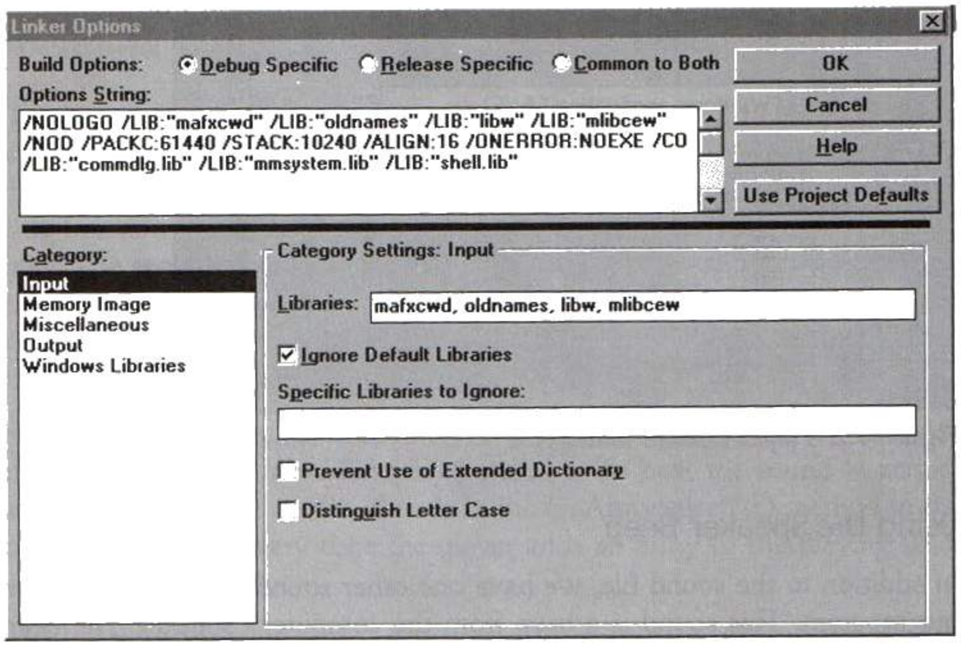
```
MessageBeep(-1);
```

By placing this statement anywhere in your code, a beep will be heard over the speaker when the code is executed. In Chapter 9, we will use this statement to warn us when a message is placed in the message list box on the operations window.

Menu Sound Select

With sound added to our game we have to consider the situation when a player does not want to hear sound. An option should be available that allows the player to turn off sound or to turn it back on once turned off. We will add this option in the menu of our game.

We have already seen how to activate a menu option when we added the Buy Equipment dialog box. The sound system will use the same type of



Build Options: [✓ Debug Specific] Release Specific Common to Both

Options String:

```
/NOLOGO /LIB:"mafxcwd" /LIB:"oldnames" /LIB:"libw" /LIB:"mlibcew"
/NOD /PACKC:61440 /STACK:10240 /ALIGN:16 /ONERROR:NOEXE /CO
/LIB:"commdlg.lib" /LIB:"mmsystem.lib" /LIB:"shell.lib"
```

Category: [✓ Input] Memory Image, Miscellaneous, Output, Windows Libraries

Category Settings: Input

Libraries: mafxcwd, oldnames, libw, mlibcew

☑ Ignore Default Libraries

Specific Libraries to Ignore:

☐ Prevent Use of Extended Dictionary

☐ Distinguish Letter Case

Figure 7.3 *Linker Options dialog box.*

menu option, except a dialog box will not be displayed and a check mark will appear next to the menu option.

Bring up AppStudio and click on Menu in the left list box. Double-click on the menu listed in the right list box. Click on the Help menu item and drag it to the end of the menu so that the blank item is between Edit and Help.

Click on the Blank entry and press return. Type in the name "options". Click on the blank entry under Options and press return. Enter the name "sound". Close the menu and click on the Resource menu item of AppStudio and then ClassWizard.

Make sure that the class CNetwar2View is in the Class Name edit control. Move through the Object IDs list box until you find the entry for ID_OPTIONS_SOUND. Click on the entry to highlight it. You will find two entries in the right-most list box. Click on the first one and click Add Function. Do the same for the second entry. Click the OK button. Save and close AppStudio.

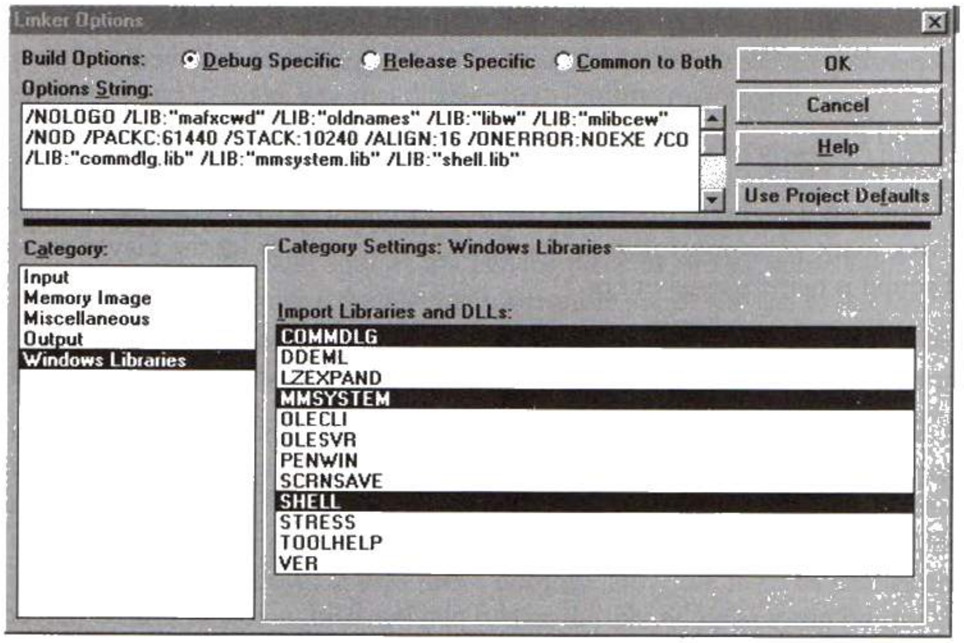
What we have just done is add a menu item to our game called Options. This menu item is a popup menu, which means that there are other menu items in a menu when Options is clicked. A menu item called Sound was added. Two functions were created by ClassWizard to service the Sound menu item. The function called OnOptionsSound() is where we will put the code to be executed when the Sound menu item is clicked.

The second function, called OnUpdateOptionsSound(), is used to place or remove a small check mark beside the menu option Sound. If a check mark is present, then all sounds will be played in the game. If a check mark is not present, no sound will be played. This menu option will work as a toggle between turning sound on and off.

Sound Variable

To determine whether or not sound is to be played, we will use a Boolean variable called *do_sound*. In the netwavw.h file, add the statement:

```
BOOL do_sound;
```



Build Options: [✓ Debug Specific] Release Specific Common to Both

Options String:

```
/NOLOGO /LIB:"mafxcwd" /LIB:"oldnames" /LIB:"libw" /LIB:"mlibcew"
/NOD /PACKC:61440 /STACK:10240 /ALIGN:16 /ONERROR:NOEXE /CO
/LIB:"commdlg.lib" /LIB:"mmsystem.lib" /LIB:"shell.lib"
```

Category: Input, Memory Image, Miscellaneous, Output, [✓ Windows Libraries]

Category Settings: Windows Libraries

Import Libraries and DLLs: [✓ COMMDLG], DDEML, LZEXPAND, [✓ MMSYSTEM], OLECLI, OLESVR, PENWIN, SCRNSAVE, [✓ SHELL], STRESS, TOOLHELP, VER

Figure 7.4 *Clicking on windows libraries.*

to the first PRIVATE area.

When the variable *do_sound* is TRUE, it indicates that all sound should be played in our game. Open the netwavw.cpp file and add the code

```
do_sound = FALSE;
```

to the view's constructor.

Move down to the OnOptionsSound and add the following code to the function.

```
if (do_sound)
   do_sound = FALSE;
else
   do_sound = TRUE;
```

In the OnUpdateOptionsSound() function, add the code

```
pCmdUI->SetCheck(do_sound);
```

The method SetCheck() places a check mark next to a menu item if its parameter is TRUE and removes a check mark if the parameter is FALSE. The value of the *do_sound* variable is sent to the function as the parameter. Depending on the value, the check mark will be placed or removed.

Sound Play

Up to this point, we have created a variable called *do_sound*. This variable will be toggled TRUE/FALSE when the Sound menu item is clicked. A small check mark will appear next to the Sound menu item to let the player know if sound is being played or not.

Now we have to use the *do_sound* variable to determine if a sound is played or not. Move to the OnSelectedArmy() function and put the following code around the two statements that play our sound:

```
if (do_sound)
{
 code here!!
}
```

When an army is to be placed on the playing field, a sound will be played only when the variable *do_sound* is TRUE; otherwise the sound will not play.

You will want to use this type of code for all sound that you have in your final game.

Sound Code

All of the code up to this point can be found in the /netwar2/chapt7/b directory.

Limiting Movement

If you have played with the versions of the game up to this point, you will have noticed that the playing pieces can be moved all over the playing field just as fast as you can move the mouse. If you think about two or even four players moving pieces over the board this rapidly, things might get confusing fast. In addition, should a big legion playing piece be able to move as fast as a flyer? The answer is no. We need to limit the movement of our playing pieces.

One way we could limit the movement of our playing pieces is to change the speed of the mouse depending on the type of playing piece being used. The problem with this technique is that the mouse cannot be slowed to a slow enough speed to make any real difference in limiting movement.

The way we are going to limit playing piece movement is to give each playing piece a value that represents the farthest distance the piece can be moved per mouse click. In other words, when a player clicks on a playing piece and starts to move it, the code will only let the playing piece be moved some value *X*. If the player wants to move the playing piece more, he or she has to release the mouse button and click on the playing piece again. This might not seem like a big deal, but it does delay the movement of the playing piece effectively.

Let's look at the code that we are going to have to add to our playing pieces and the mouse movement handlers in the view object. When a playing piece is purchased and added to the system in the buy_equipment method, a speed value is assigned as the last parameter sent to the sprite initialization code. Up to this point, this value was just a dummy. Now we need to change the value to indicate the number of pixels that the playing piece can be moved per mouse click. The values that we are going to use are

flyer	=	90
archer	=	30
cavalry	=	40
legion	=	20

You can change these numbers to anything you might like, just remember that they should be proportionate. Open the netwavw.cpp file and put these numbers in their appropriate place in the buy_equipment method.

Now we need to turn our attention to how we are going to determine the number of pixels the mouse moved after it has clicked on a playing piece. The method that does the actual movement of the playing piece is called OnMouseMove(), and you can find it in the netwavw.cpp file. Right now all the function does is check to make sure that we are dragging a playing piece and call the MoveTo() sprite method to move the playing piece to the current location of the mouse.

That's it. In this function, we have the current location of the mouse in the *point* variable sent as a parameter to this function. The length of a line can be calculated relatively easily, as long as we know the starting point and the ending point. We have the ending point, so what about the starting point?

When the player clicks on the playing piece to move, we have a starting point. All that needs to be done is to remember this starting point. That's easy, since we have objects. Open the sprite.h file and add the following code to the first PUBLIC area.

```
int startx, starty;
```

These variables will be used to hold the starting X and starting Y location of the playing piece to be moved when the player first clicks on it. To put values into these variables, we need to move to the OnLButtonDown() method in the netwavw.cpp file.

In this method, add the following code in the middle of the IF {} statement.

```
playing_pieces[who_am_i][i]->startx = point.x;
playing_pieces[who_am_i][i]->starty = point.y;
```

These statements take the current position of the mouse and place its value into the *startx* and *starty* member variables for the playing piece clicked on. This gives us the starting point of a line. When the user moves the mouse, we will have an end point for the line. Now we use the equation for the distance of a line to find out how many pixels the playing piece is going to be moved. If the number of pixels is greater than the speed value for this playing piece, then the playing piece is not moved.

All of this takes place in the OnMouseMove() function. Replace the code in the function so that it looks like this:

```
void CNetwar2View::OnMouseMove(UINT nFlags, CPoint point)
{
   int move_x, move_y, length;
   if (dragging == -1)

      return;

   move_x = (point.x - playing_pieces[who_am_i][dragging]->XOffset);
   move_y = (point.y - playing_pieces[who_am_i][dragging]->YOffset);

   length = (int)sqrt(pow((double)point.x-playing_pieces[who_am_i][dragging]->startx,
2) + pow((double)point.y-playing_pieces[who_am_i][dragging]->starty, 2));

   if ((length < playing_pieces[who_am_i][dragging]->speed))
   {
    playing_pieces[who_am_i][dragging]->MoveTo(GetDC(), move_x, move_y);
   }
}
```

To compile this code, you will need to add the *include* statement:

```
#include "math.h"
```

to the top of the netwavw.cpp file.

As you can see from the code above, it does everything we talked about. It first calculates the length of the line from the original starting point of the playing piece to the current location of the mouse cursor. If this length is less than the allowed movement speed of the current playing piece, the playing piece is moved to the correct position; otherwise nothing happens to the piece.

Cost Per Move

During a real battle, there is always a cost involved in moving an army from one position to another. The same will be true in our game. With just four lines of code, we can assess a gold cost to each movement of a playing piece. As far as game design goes, the movement cost adds a great deal of strategy. Players cannot continuously move their units around the playing field. Players must choose their maneuvers and battles carefully—if a player runs out of gold, his or her armies will be permanently immobilized, becoming easy prey for the enemy's archer units.

Open the netwavw.cpp file and find the OnMouseMove() method. At the top of the function, add a declaration

```
int cost;
```

This variable will be used to calculate the total number of gold coins a playing piece has used during its current movement.

Add the following code under the calculation of the length of the playing piece move.

```
cost = length * playing_pieces[who_am_i][dragging]->speed;
```

The total cost of the playing piece move is the length of the move times the speed factor for this particular playing piece. This is obviously an area that you can change to anything you would like. Now we need to assess the cost to the playing piece. We only want to assess the cost when the playing piece is actually moved, but we also only want to assess the cost when the player has enough money to make the move. This means that in order to move a playing piece, a player must have enough money to make the move.

Change the IF statement in the function to read:

```
if ((length < playing_pieces[who_am_i][dragging]->speed) && (cost < players[0]-
>gold_coins))
{
   players[0]->gold_coins -= cost;
   ops_dlg->total_gold = players[0]->gold_coins;
   ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);
```

```
    playing_pieces[who_am_i][dragging]->MoveTo(GetDC(), move_x, move_y);
}
```

The code checks to make sure that the playing piece isn't being moved outside its speed parameter and the player has enough money for the move. If everything looks good, the cost of the move is subtracted from the player's gold coins. Next, the player's new gold count is displayed in the operations window and the playing piece is moved.

New Game

The last thing we are going to add to our game before multiplayer support is a menu option for a new game. The option for a new game will be under the menu item File. Open AppStudio and select the Menu resource. Double-click on our menu in the right-hand list box.

Click on the File menu item. Double-click on the entry for New. In the Caption edit box, delete the current name and enter "New Game". Close the properties dialog box and close the menu. Now click on Resources and ClassWizard. Make sure the CNetwar2View is the current class. Find the Object ID called ID_FILE_NEW. Click to highlight this option and click on the first entry that appears in the right list box. Click on the Add Function button to add a handler for this menu item.

Close ClassWizard and AppStudio. Open the netwavw.cpp file and find the method OnFileNew(). Replace the code in this function with the following:

```
int i;
 ops_dlg->SendMessage(WM_OPERATIONS_NEW_GAME, 0, 0L);
 for (i=0;i<total_pieces[0];i++)
    delete playing_pieces[0][i];

 total_pieces[0] = 0;

 Invalidate(TRUE);
 the_players[0]->gold_coins = 500000;
 the_players[0]->kills = 0;
```

The code is fairly straightforward. First, a message is sent to the operations window class letting it know that it should reset all of its controls. We will have to look at this code in a minute. Next, all of the playing pieces for our player are destroyed and the screen is updated.

Last, the gold coins for our player are put back to 500,000 and the number of kills is set to 0. Everything will be ready for a new game at this point.

Operations Window Reset

The one thing we didn't look at was resetting the operations window. Remember that we have text in the Armies list box and may have text in the message list box. By sending a message to the operations window, we can allow it to reset itself instead of having the view object do it.

In the operatio.h file, add the message definition code to the top of the file.

```
#define WM_OPERATIONS_NEW_GAME WM_USER + 105
```

Later in the file, add the message handler method prototype.

```
long OnNewGame(UINT wParam, LONG lParam);
```

Now open the operatio.cpp file and create a message macro for the message map.

```
ON_MESSAGE(WM_OPERATIONS_NEW_GAME, OnNewGame);
```

Move to the bottom of the file and add the handler.

```
long operations::OnNewGame(UINT wParam, LONG lParam)
{
   CListBox* box;
   CEdit* box2;

   box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
   box->ResetContent();

   box = (CListBox*)GetDlgItem(IDC_OPERATIONS_MESSAGE_BOX);
   box->ResetContent();

   box2 = (CEdit*)GetDlgItem(IDC_OPERATIONS_AVAILABLE_GOLD);
   box2->SetSel(0, -1, FALSE);
   box2->ReplaceSel("");
   total_armies = 0;
   return 1;
}
```

The new game handler does three tasks. First, it removes all of the items in the Armies list box. Second, it removes all messages in the message list box. Third, it resets the value in the available gold edit control to a blank.

The code for this chapter can be found in the netwar2/chapt7/c directory.

Summary

With the addition of the war engine, sound effects, and new game option, a complete single-player game has been created. The game is very limited, however, since it was designed to be played against a single opponent.The remaining chapters in the book will explore the different network options available for adding opponents to our game.

Chapter 8

Making Connections Using Sockets

With every operating system developed for research and commercial use, a different scheme is devised for communicating between processes. Some of these operating systems are designed in such a way that two processes executing on the same machine might not be able to communicate with each other because certain assumptions have not be met. When the UNIX operating system was being designed, there was a call for a common communication mechanism. This is where sockets came into play.

Berkeley Sockets

Sockets were developed to give UNIX a standard protocol for interprocess communication. Along with providing a standard protocol, sockets allow the processes to communicate using several different protocols including UDP and TCP/IP. Figure 8.1 gives a layered look at sockets.

As you can see, process A wishes to communicate with process B. Process A executes socket functions to start the communication. The socket layer translates the socket functions into systems calls at lower levels in the operating system. The information to be sent to process B enters the protocol layer, where packets are created for the information. The packets will be in the form of UDP or TCP/IP. These packets are transferred to the device layer, where a software driver for an ethernet card in the computer sends the packets out on the network to the machine with process B executing on it.

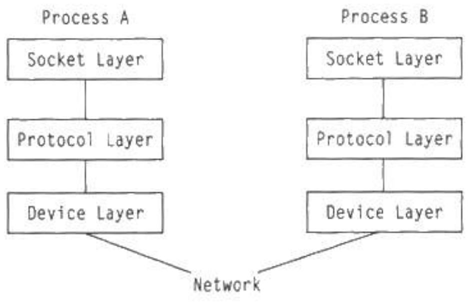


Figure 8.1 *Layered look at sockets.*

When the information arrives at process B, the reverse sequence of layers occurs. The packets are taken from the network card and translated from TCP/IP or UDP into the data sent from process A. Socket function calls on process B requesting information retrieve the information for use.

The communication connection can be one of two different types. The first is called a stream. A stream is a sequenced and reliable delivery mechanism. The second is called a datagram. A datagram does not guarantee delivery, sequence, or unduplicated packets, but it is less expensive.

When connecting, processes act in a client/server protocol. The server listens for connections, and the client initiates them. Sockets and the connection between them are initiated with the command socket(). A typical function call is

```
socket_descriptor = socket(format, type, protocol);
```

In this function call, the parameter *format* specifies the domain to use in the communication. The domain can be either UNIX or Internet. The UNIX domain format is used when the communicating processes are on the same machine. The *type* parameter describes the type of communication, stream or datagram. The *protocol* parameter indicates the protocol that should be used for all communication. This is TCP/IP or some other protocol. Once a socket has been created, its descriptor will be used in many other socket functions.

One of these functions is bind. Bind() associates a name with a socket. Typically the name is the name of the machine or Internet address that we wish to connect. The function call is

```
bind(socket_descriptor, address, length);
```

The socket_descriptor is the socket to use in the bind. *Address* is a pointer to a structure that specifies information about the machine to connect. The parameter *length* is the length of the structure in the second parameter. The bind() function call is used by the server in the connection. The bind() function will advertise the name of this socket to others and allow socket connections to occur.

Once a server has issued a bind() function call, it will follow with listen(). The function call appears as

```
listen(socket_descriptor, number);
```

The first parameter is obvious. The second parameter, *number,* is a value for the number of connections that can be queued with this server. This does not mean that there will be this number of connections, but only that this number of connections can be queued.

The server process will listen for connections from clients. When a connection has been detected, the function connect() will be executed. The call is

```
connect(socket_descriptor, address, length);
```

This function call's second parameter is the server's information to which we wish to connect. The information in this structure will be much the same as in the structure of the server. Once the connect() and listen() function calls are executed, a connection has to be created with the accept() function. This function call is made by the server

```
new_socket_descriptor = accept(socket_descriptor, address, length);
```

The function creates a connection with the client using the socket *socket_descriptor*. Information about the client will be placed in the *address* structure. The parameter *length* is the length of the structure in the second parameter. Once the accept() function call has been executed, a new socket descriptor is created and returned. This is the socket that will be used to communicate with the client, not the old *socket_descriptor.*

With a connection created, the processes can send and receive information. Sending information is performed with the command send(). The call is

```
character_count = send(new_socket_descriptor, message, length, flag);
```

The function call sends the character information in the *message* parameter to the process connected to the socket *socket_descriptor*. The size of the message to be sent is placed in the third parameter, *length*. A flag can be sent, but will typically be zero. After the function call has been executed, the number of characters sent is returned. The receiving machine issues the command

```
count = recv(new_socket_descriptor, buffer, length, flags);
```

to receive the information sent by the other process. The second parameter is a buffer for the incoming data. The function returns the total number of characters put into the buffer. The third parameter is the size of the buffer. It should be noted that the functions read() and write() can be used in place of send() and recv().

After all is said and done, the socket can be closed with the command

```
shutdown(new_socket_descriptor);
```

Sample Code

Now we will look at some example code for a server and a client application. The server code is

```
#include <sys/types.h>
#include <sys/socket.h>

main()
{
 int sd, nsd;
 char buffer[256];
 struct sockaddr sockaddr;
 int len;

 sd = socket(AF_UNIX, SOCK_STREAM, 0);
 bind(sd, "sock", sizeof("sock"), -1);
 listen(sd, 1);

 for(;;)
{
  nsd = accept(sd, &sockaddr, &len);
  if (fork() == 0)
  {
  close(sd);
  read(nsd, buffer, sizeof(buffer);
  printf ("server received: %s\n", buffer);
  exit();
  }
close(nsd);
  }
}
```

The code starts by creating a socket using the socket() function. The domain used is AF_UNIX, which means that the communication can only be between processes on a single machine. The type of communication is SOCK_STREAM for reliable communication. The socket is bound to the name "sock". The server is told to listen for a single connection.

At this point, an endless loop is entered to process all client connections. The first statement in the loop is the accept() function. Once a connection is created, a process is forked. The new process reads a character string from the client, prints the string, and exits the process.

Now for the client code—

```
#include <sys/types.h>
#include <sys/socket.h>

main()
{
   int sd, nsd;
   char buffer[256];
   struct sockaddr sockaddr;
   int len;

   sd = socket(AF_UNIX, SOCK_STREAM, 0);

   if (connect(sd, "sock", sizeof("sock"), -1) == -1)
      exit();
   write (sd, "HI Mister Server!!", 16);
}
```

That's all there is to a socket connection. All of this code so far has been in the realm of a UNIX system. We are using Microsoft Windows for our game, so how are we going to use sockets?

Microsoft Winsock

When Microsoft Windows became a driving force in the operating system war for the PC, developers saw the need to connect with UNIX boxes. The developers set out to create a socket system for Microsoft Windows. The system is called Winsock. The current specification is 1.1, with 2.0 coming in mid-to-late 1996.

The purpose of Winsock is to create a single API from which developers can create socket applications that will communicate with both UNIX and Windows applications. The specification in Appendix C goes into great detail about the workings of Winsock. In this section, we will pinpoint one aspect.

UNIX is a true multitasking operating system. Microsoft Windows is not. For this reason, Winsock has to treat the accept() command in the server differently than in UNIX. If you look at the preceding code, you will find that the code waits at the *accept()* statement until a connection is available from a client. If a Windows application were to do this, the application would take complete control over the system. In Windows, an application has to give up control to others. To execute a function that does not return for potentially a long time, a different solution is necessary. The solution is a new accept() function that returns a message to the application when a client is trying to make a connection. Let's see how the code for sockets looks in Winsock compared to the code in UNIX presented earlier.

A Simple Communication Program Using Sockets

In order to show how to use the Winsock system, we will design a simple application that allows a client to connect to a server. Begin by creating an application using AppWizard. Go into AppStudio and add two menu items to the current menu. The names of the menu items are Client and Server. These are menu items and not popups, so make sure you click off the check mark in the popup radio box. Move to ClassWizard and add message handler functions to the program.

In the header file for the program, add the following code to the first PUBLIC heading.

```
WSADATA             wsaData;
int                 a_socket;
struct sockaddr_in  connection;
```

```
int                  client;
struct sockaddr_in   client_addr;
```

These are the variables necessary for the sockets and connection between the client and server. The breakdown of the variables is as follows:

wsaData	This variable is used to load the Winsock DLL into the Windows system.
a_socket	This variable will be used by the server to create a socket which accepts connections from a client. The client will use the variable as a socket to connect with the server.
connection	This variable is used to hold information about the type of connection that will be created between the client and server.
client	This variable is used by the server to create a final socket with the client.
client_addr	This variable is used by the server to hold connection information about the client.

In the header file you will also need to add the following include file:

```
#include "winsock.h"
```

This file contains all of the necessary constants and prototypes for the Winsock system.

Server

Now open the source file for our program and add the following code to the OnServer () function.

```
WORD wVersionRequested;
 int err, len_connection_addr;

 wVersionRequested = 0x0101;
 err = WSAStartup(wVersionRequested, &wsaData);
 if (err != 0)
 {
  MessageBox("Winsock Version 1.1 not found...", "No Network", MB_OK | MB_ICONSTOP);
  return;
 }
```

```
else
{
 MessageBox("Winsock found...OK for use", "Found Network", MB_OK);
}

a_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (a_socket < 0)
{
 MessageBox("Unable to create socket", "No Socket", MB_OK);
 return;
}

len_connection_addr = sizeof(struct sockaddr_in);
connection.sin_family = AF_INET;
connection.sin_addr.s_addr = htonl(INADDR_ANY);
connection.sin_port = htons(5002);

if (bind(a_socket,(struct sockaddr *)&connection, len_connection_addr) < 0)
{
   MessageBox("Unable to bind socket", "No Bind", MB_OK);
   return;
}

listen(a_socket, 3);

WSAAsyncSelect(a_socket, m_hWnd, MSG_ACCEPT, FD_ACCEPT);
```

The code for the server starts by trying to load the Winsock DLL into the Windows system. If the Winsock is loaded currently, a message box is displayed indicating a good load. If there are problems with the DLL and its loading, a different message box is displayed and control is returned from the function.

Once the DLL is loaded, a socket is created using the command socket(). The parameters for the socket() function indicate that we are going to be connecting through the Internet using a reliable connection and a TCP/IP protocol. If the socket is not able to be created, a message box is displayed. Otherwise, the code continues.

The code fills in the *connection* structure next.

```
len_connection_addr = sizeof(struct sockaddr_in);
connection.sin_family = AF_INET;
connection.sin_addr.s_addr = htonl(INADDR_ANY);
connection.sin_port = htons(5002);
```

The first line of code sets a local variable called *len_connection_addr* equal to the size of the *connection* structure. The second line of code sets the *sin_family* field equal to AF_INET. This indicates to the connection that we will be using the Internet as our connection medium. The third line of code sets the field *sin_addr.s_addr* equal to the constant INADDR_ANY. This tells the connection that any Internet address can connect to the server. The fourth line of code sets the field *sin_port* equal to a port value. The port value is like a room at some address. The address is the Internet address of the machine.

The next bit of code performs the bind() operations discussed above.

```
if (bind(a_socket,(struct sockaddr *)&connection, len_connection_addr) < 0)
{
   MessageBox("Unable to bind socket", "No Bind", MB_OK);
   return;
}
```

The socket created in the variable *a_socket* is bound to the connection information in the variable *connection*. If the bind is not successful, a message box is displayed. Now the server has to listen for connections. The code

```
listen(a_socket, 1);
```

tells the system to queue up to 1 connection. The last line of code is where the socket code changes from UNIX to Windows. In the UNIX version of the server socket code, the function accept() would be executed. In Windows, we cannot execute a single statement and wait for the connection from the client. What Winsock does is register an accept() function so that when a client tries to connect to the server, a message is sent to the application. The code is

```
WSAAsyncSelect(a_socket, m_hWnd, MSG_ACCEPT, FD_ACCEPT);
```

Now we need to create a message macro for the message MSG_ACCEPT. The code is

```
ON_MESSAGE(MSG_ACCEPT, OnClientConnect)
```

Now we add the message handler code

```
long COnecommView::OnClientConnect(UINT wParam, LONG lParam)
{
  int len;

  len = sizeof(struct sockaddr_in);
  client = accept(a_socket, (struct sockaddr *)&client_addr, &len);
  if (client < 0)
 {
  MessageBox("Error in Accept with client", "Error", MB_OK);
  return 0;
 }

 WSAAsyncSelect(client, m_hWnd, MSG_FROM_CLIENT, FD_READ);

 send(client, "Hi", 2, 0);

 WSAAsyncSelect(a_socket, m_hWnd, 0, 0);

 return 1;
}
```

When a client tries to connect with the server, the message MSG_ACCEPT is sent to the program. The program in turn executes the code in the OnClient-Connect() function. This function starts by issuing the accept() socket function. The return value of the accept() command is the socket to use when communicating with the client. This return value is stored in the *client* variable for later use. After the connection is established, the code registers the new socket along with the flag FD_READ and the message MSG_FROM_CLIENT. This registering tells Winsock that when information is received by the socket *client*, it should send a message to the program called MSG_FROM_CLIENT.

For our testing, we follow the registering with the function send(). We send a string called "Hi" to the client to verify the connection. The next to the last line of code tells Winsock that the first registering we did was with the *a_socket* socket and the accept flag should be canceled. This means that no other clients will be allowed to connect with the server.

When information is received from the client, the message MSG_FROM_CLIENT is sent to the program. We need a message macro like

```
ON_MESSAGE(MSG_FROM_CLIENT, OnMsgFromClient)
```

The code for this message handler is

```
long COnecommView::OnMsgFromClient(UINT wParam, LONG lParam)
{
  int count;
  char buffer[256];

  count = recv(client, buffer, 256, 0);
  buffer[count] = '\0';

  MessageBox(buffer, "From Client", MB_OK);

  return 1;
}
```

This function reads information from the *client* socket and displays the information in a message box.

Everything is now set on the server side. To review, when the player clicks on the Server menu item, the code reads in the Winsock DLL, creates a socket to receive connections from a client and registers an accept flag with the Winsock system. When a client tries to connect to the server, the message MSG_ACCEPT is sent to the program. The message causes the code OnClientConnect() to be executed. This code creates a new socket with the client and sends some information to the client. The original accept registration is canceled.

Client

The client connects to the server when the Client menu item is clicked. This causes the code in the OnClient() function to be executed. This code is:

```
WORD wVersionRequested;
   int err, len_connection_addr;

   wVersionRequested = 0x0101;
   err = WSAStartup(wVersionRequested, &wsaData);
   if (err != 0)
   {
     MessageBox("Winsock Version 1.1 not found...", "No Network", MB_OK | MB_ICON-
STOP);
     return;
   }
   else
   {
     MessageBox("Winsock found...OK for use", "Found Network", MB_OK);
   }
```

```
  len_connection_addr = sizeof(struct sockaddr_in);
  connection.sin_family = AF_INET;
  connection.sin_addr.s_addr = inet_addr("129.72.6.194");
  connection.sin_port = htons(5002);
  a_socket = socket(AF_INET, SOCK_STREAM, 0);

 if (a_socket < 0)
 {
  MessageBox("Unable to create socket", "No Socket", MB_OK);
  return;
 }

 if (connect(a_socket, (struct sockaddr *)&connection, len_connection_addr) < 0)
 {
  MessageBox("Unable to create a connection to server", "No Connection", MB_OK);
  return;
 }

 WSAAsyncSelect(a_socket, m_hWnd, SERVER_MSG, FD_READ);

 send(a_socket, "Hello", 5, 0);
```

The code for the client starts out like the server by loading the Winsock DLL. After the DLL is loaded, the client loads a connection structure with necessary information

```
 connection.sin_family = AF_INET;
 connection.sin_addr.s_addr = inet_addr("129.72.6.194");
 connection.sin_port = htons(5002);
```

The only difference in this loading code and the server's loading code is the second line. The client must know the IP address of the server. In testing this code, I used a machine at the address "129.72.6.194".

Once the connection structure is created, a socket is created to connect with the server

```
a_socket = socket(AF_INET, SOCK_STREAM, 0);
```

After the socket is created, a connection is tried with the server with the code:

```
if (connect(a_socket, (struct sockaddr *)&connection, len_connection_addr) < 0)
 {
  MessageBox("Unable to create a connection to server", "No Connection", MB_OK);
  return;
 }
```

If the connection is not successful, a message box is displayed. Otherwise, the code registers with the Winsock DLL to send the message SERVER_MSG when the client receives any information from the server.

```
WSAAsyncSelect(a_socket, m_hWnd, SERVER_MSG, FD_READ);
```

The last line of code sends a simple message to the server after the connection has been made.

```
send(a_socket, "Hello", 5, 0);
```

We need to add a message macro and message handler for the SERVER_MSG. The macro is

```
ON_MESSAGE(SERVER_MSG, OnServerMsg)
```

The message handler is

```
long COnecommView::OnServerMsg(UINT wParam, LONG lParam)
{
 int count;
 char buffer[256];

 count = recv(a_socket, buffer, 256, 0);
 buffer[count] = '\0';

 MessageBox(buffer, "From Server", MB_OK);

 return 1;
}
```

As in the case of the server, the client receives information from the server and displays it in a message box.

Test

Compile the project and execute it on two different machines connected on the Internet. Make sure that you place the correct IP address in the client function. On the server machine, click the Server menu item. A message should be displayed letting the user know Winsock has been loaded. Click OK. The server will be waiting for a connection.

If the connection is not successful, a message box is displayed. Otherwise, the code registers with the Winsock DLL to send the message SERVER_MSG when the client receives any information from the server.

```
WSAAsyncSelect(a_socket, m_hWnd, SERVER_MSG, FD_READ);
```

The last line of code sends a simple message to the server after the connection has been made.

```
send(a_socket, "Hello", 5, 0);
```

We need to add a message macro and message handler for the SERVER_MSG. The macro is

```
ON_MESSAGE(SERVER_MSG, OnServerMsg)
```

The message handler is

```
long COnecommView::OnServerMsg(UINT wParam, LONG lParam)
{
  int count;
  char buffer[256];

  count = recv(a_socket, buffer, 256, 0);
  buffer[count] = '\0';

  MessageBox(buffer, "From Server", MB_OK);

  return 1;
}
```

As in the case of the server, the client receives information from the server and displays it in a message box.

Test

Compile the project and execute it on two different machines connected on the Internet. Make sure that you place the correct IP address in the client function. On the server machine, click the Server menu item. A message should be displayed letting the user know Winsock has been loaded. Click OK. The server will be waiting for a connection.

On the client machine do the same sequence except click on Client. When the Winsock message box is displayed, click on OK. At this point, message boxes will be displayed on each of the client and server machines.

The code for this program can be found in the netwar2/chapt8/a directory.

Client/Server Communications

We can expand the preceding program to allow more than one client to attach to the server. In the header file, change the declarations from

```
int                   client;
struct sockaddr_in    client_addr;
```

to

```
int                   client[2];
struct sockaddr_in    client_addr[2];
```

and add the declaration

```
int connections;
```

We are going to allow up to two clients to attach to the server. Now open the source file and change the OnClientConnect() function to read

```
int len;

  len = sizeof(struct sockaddr_in);
  client[connections] = accept(a_socket, (struct sockaddr *)&client_addr[connections],
&len);
  if (client[connections] < 0)
  {
    MessageBox("Error in Accept with client", "Error", MB_OK);
    return 0;
  }

  if (connections == 0)
  {
    WSAAsyncSelect(client[0], m_hWnd, MSG_FROM_CLIENT, FD_READ);
    send(client[0], "Hi", 2, 0);
  }
  else
  {
   WSAAsyncSelect(client[1], m_hWnd, MSG_FROM_CLIENT2, FD_READ);
   send(client[1], "Hi", 2, 0);
  }

  connections++;

  if (connections == 2)
   WSAAsyncSelect(a_socket, m_hWnd, 0, 0);

  return 1;
```

Notice that the code starts with the same accept() function, but the result of the function is assigned to the *client[connections]* variable and the structure used in the accept function is also one of the positions in an array of connection structures.

We need to make sure that the variable *connection* is set to zero. So move up to the constructor function and add the code

```
connections = 0;
```

As you move down in the earlier code, you will see that depending on which client has been connected to the server, a different message is sent to the server when information is sent from that client. In this case, we have added a message called MSG_FROM_CLIENT2 and a new message handler. The code is

```
long COnecommView::OnMsgFromClient(UINT wParam, LONG lParam)
{
 int count;
 char buffer[256];

 count = recv(client[0], buffer, 256, 0);
 buffer[count] = '\0';

 MessageBox(buffer, "From Client 0", MB_OK);

 return 1;
}


long COnecommView::OnMsgFromClient2(UINT wParam, LONG lParam)
{
 int count;
 char buffer[256];
 count = recv(client[1], buffer, 256, 0);
 buffer[count] = '\0';

 MessageBox(buffer, "From Client 1", MB_OK);

 return 1;
}
```

Now add an additional message macro

```
ON_MESSAGE(MSG_FROM_CLIENT2, OnMsgFromClient2)
```

That's all of the necessary code for multiple clients. All we need to do is have an array of sockets for each of the clients and connection structures. A separate message and handler are called for each client that sends information.

Test

The code for multiple clients can be found in the /netwar2/chapt8/b directory. Start the server and client on Server and the message box OK. Then start the code on two additional machines and click on Client for each. They will connect to the server and display appropriate messages.

Summary

In this chapter we have discussed the development of sockets and their migration to Microsoft Windows. Using the library and DLL for Winsock, a simple communication program was developed that allowed any two machines on the Internet with valid IP addresses to communicate with each other. This communication program was expanded to a client/server protocol. All of this information will be used in the next chapter to add sockets and Internet communication to our game.

Chapter 9

Socket Support for the Game

The last two chapters have outlined the details and techniques necessary for using sockets and specifically Windows Winsock for Internet communications. With all of this knowledge at our disposal, we are ready to incorporate Winsock into our game so that multiple kings can try to destroy each other. After reading this chapter and entering the code, or examining it on the enclosed CD-ROM, you will have a complete multiplayer game that allows up to four players to play over the Internet.

Determining Server and Clients

As you will recall, the underlying structure for multiplayer play in our game will be based on a client/server model. One of the machines in the game will be designated as a server; all others will be clients. The server will wait for all of the clients to register with it before the game can begin. Since we are using the client/server model, each player will need to know which of the machines is the server before trying to connect. All the players will need is the IP address of the server machine.

In order to keep things straight and initialize the game correctly, all players will be required to enter information about themselves and possibly the server in the game. When players first begin the game, they will click on a menu item called Players. Under Players will be the entry for Enter Players.

Clicking this menu item will bring up the dialog box in Figure 9.1. This dialog box is asking for the player to enter information about him- or herself.

The two most important things on this dialog box are the client/server buttons and the number of players edit control. The name of the player is something added but not used in this version of the game. The player is required to click on one of the Client/Server buttons and enter the total number of players in the game. The total number of players must be between 2 and 4. An error dialog box will appear if the player enters a number outside of this range.

If the player clicks on the Server button, the system will automatically set things up so that this player is the server and all other players are clients. If the player clicks on the Client button, the system will display the dialog box in Figure 9.2.

The new dialog box asks for the server's name, which is not very important, and Internet address or telephone number. This player is now ready to connect with the server and start the game.

Input Self Dialog Box

The first step in determining players is to design the input player dialog boxes. There are two input dialog boxes, one for the player and one for input of server information if needed. We will design the player dialog box first.

Open AppStudio and select the option for a new dialog box. The dialog box that we are creating should appear as in Figure 9.3. The particulars are

dialog ID	=	IDD_INPUT_SELF
Client button ID	=	IDC_RADIO_CLIENT

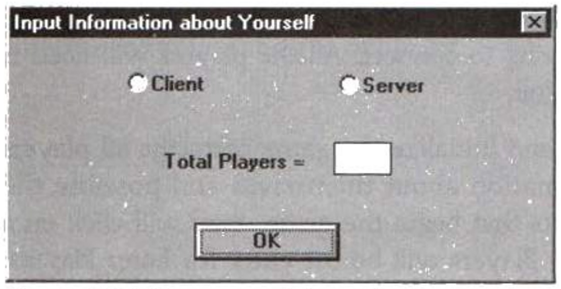


Figure 9.1 *Enter your information.*

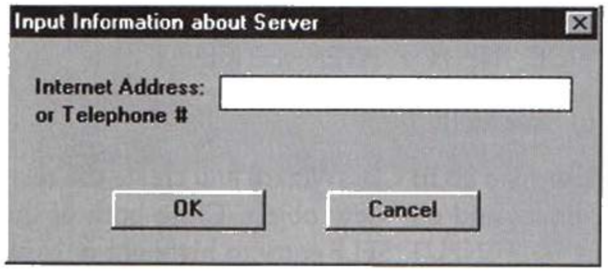


Figure 9.2 *Input information about your server.*

Server button ID	=	IDC_RADIO_SERVER
Total player edit control ID	=	IDC_INPUT_TOTAL_PLAYERS

The static text with the words "Total players = " does not need an ID. You can design the dialog box any way you want, as long as the preceding IDs are set correctly. Save this dialog box and open a new one.

Input Server Dialog Box

The new dialog box will be for entering information about the server. The dialog box should look like Figure 9.4. The particulars of this dialog box are

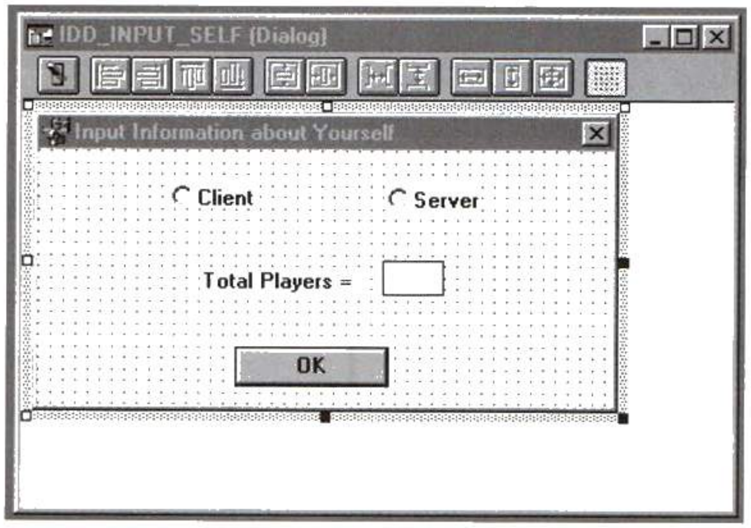


Figure 9.3 *IDD_INPUT_SELF(Dialog).*

dialog ID	=	IDD_INPUT_SERVER
address edit control	=	IDC_INPUT_PLAYER_INTERNET

The static text does not need to have an ID.

With the dialog boxes created, we can go to ClassWizard and create the relationships between the dialog boxes and our view object. Close both of the new dialog boxes. Click on the IDD_INPUT_SELF entry to highlight it. Now click on the Resources menu item and then ClassWizard.

Creating Classes with ClassWizard

You will receive a dialog box asking you to create a new class for this dialog box. Enter the name "CInputSelf" and click on Create Class. ClassWizard will create a new class for you. You should now have the dialog box shown in Figure 9.5 in front of you.

We need to create message handlers for each of the client and server buttons. Click on the entry for IDC_INPUT_CLIENT, then click on BN_CLICKED and Add Function. Do the same sequence of steps for the IDC_INPUT_SERVER ID.

Now click on Member Variables at the top of the dialog box. Since the total players edit control will contain the total number of players in our game, we

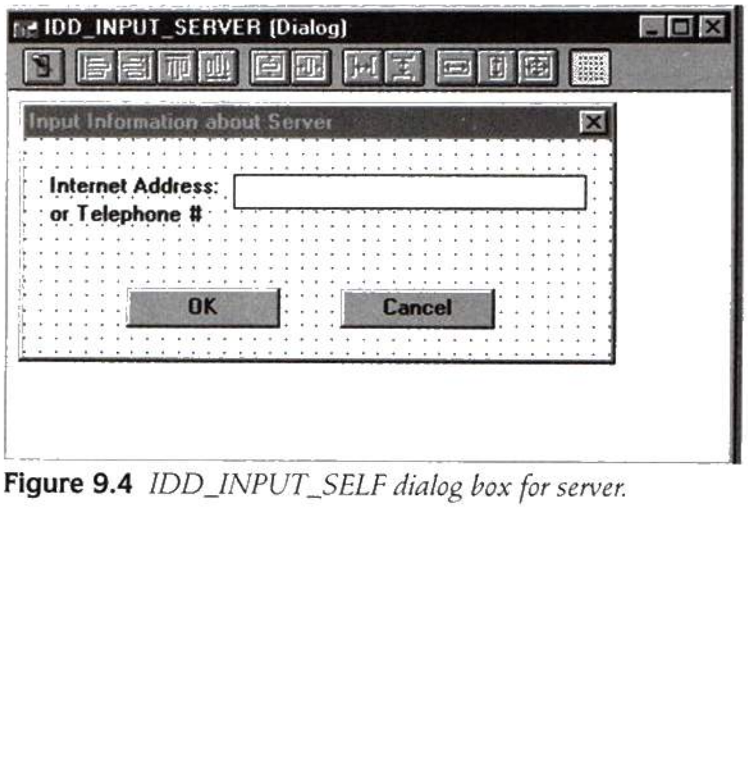


Figure 9.4 *IDD_INPUT_SELF dialog box for server.*

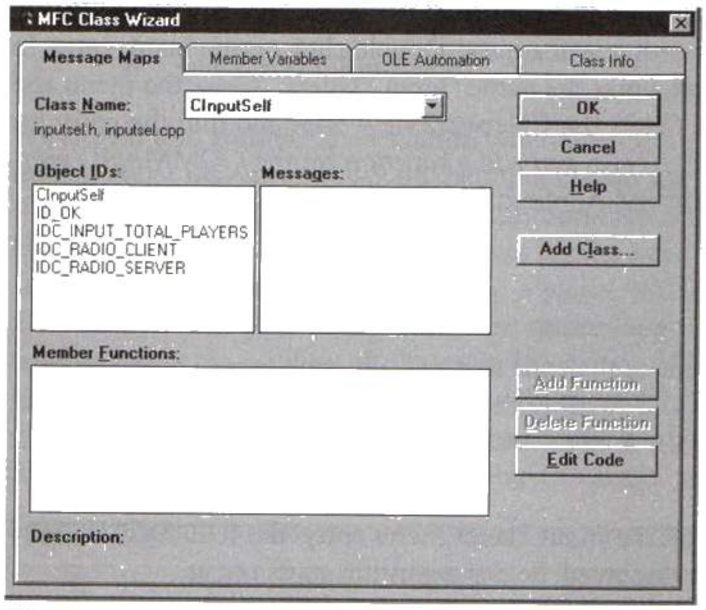


Figure 9.5 *MFC ClassWizard message maps.*

need to create a variable that we can use to access this value. Click on IDC_INPUT_TOTAL_PLAYERS and then click on Add Variable. Type in the name "m_total_players" followed by a click on OK. That's all of the controls for the IDD_INPUT_SELF dialog box, so click on the OK button to return to AppStudio.

Now click on the entry for IDC_INPUT_SERVER and click on Resources, then on ClassWizard. The only thing we need to do with this dialog box is create a member variable. First, enter the name of the new class as CInput-Server and click Create Class. Move to Member Variables. Click on IDC_INPUT_INTERNET_ADDRESS and Add Variable. Enter the name "m_input_internet". Click on OK to close ClassWizard.

Menu Option

As we noted in the description of players entering information about them-selves, the entire process starts with the player clicking on a Player's menu item and then on an Input Players entry. While we are still in AppStudio, we should add the necessary menu items.

Open our menu and click on the Players menu item. This is a popup menu, and the entries underneath it will appear. Double-click on the blank entry. In the Caption edit control, enter the name "Input Players". Close the menu and bring up ClassWizard. Select the CNetwar2View class and find the entry for ID_PLAYERS_INPUTPLAYERS and add a function for the COMMAND message. Close ClassWizard and AppStudio.

InputPlayer Code

We are now ready to create the code that will allow players to enter information about themselves and possibly the server. Before we jump into the code, however, we should take a closer look at what needs to happen during the input process.

When the user clicks on the Input Player menu entry, the IDD_INPUT_SELF dialog box needs to be displayed. So our algorithm starts out as

```
1. Display IDD_INPUT_SELF dialog box
```

The user will enter data into this dialog box and click on OK. If the player has clicked the Server button, then a SERVER object should be created (this will be discussed later) and the *who_am_i* variable should be set to 0, indicating that this is the first player in the game. In addition, some number of clients should be set up based on the number of players in the game.

If the player has clicked on the Client button, this player should be set up as a client and the IDD_INPUT_SERVER dialog box displayed. The information obtained from the IDD_INPUT_SERVER dialog box will need to be stored so that the player knows about the server.

This leads us to the algorithm

```
1. Display IDD_INPUT_SELF dialog box
2. IF player = SERVER
    3. Create server object
    4. who_am_i = 0
    5. create clients
6. ELSE player = CLIENT
```

```
7. Display IDD_INPUT_SERVER dialog box
8. Save server information
```

We should also consider the situation when users have already entered information about players and click on the Input Players menu item a second time. We don't want them to be able to enter information now, because the system has already created the necessary objects for the current game. This type of situation can be easily controlled using a global variable called *total_players*. This variable will be used by the entire system and those components that need to know the number of players in the game. When information is read into the system for the first time, we will set the value of *total_players* to the value the player entered for the total number of players. At the beginning of the code, we will check this global variable to see if it has a value other than 0.

So our algorithm is now

```
0. If total_players = 0
   1. Display IDD_INPUT_SELF dialog box
   2. IF player = SERVER
      3. Create server object
      4. who_am_i = 0
      5. create clients
   6. ELSE player = CLIENT
      7. Display IDD_INPUT_SERVER dialog box
      8. Save server information
   9. Record player count in total_players
```

Player Numbers

As the players enter information about themselves and the server, the system has to have a way of controlling and keeping track of which players are which. Remember that we have a class called Cplayer that keeps track of information about the players in the game. Each player in the game will use position 0 as his or her own, and all of the remaining positions are not necessary. Therefore you could get rid of the array of Cplayer objects if you wanted to. They will be left in the code because you could easily extend the game to keep track of all players' gold, names, or other information.

In addition to the player array, we have a variable called *who_am_i*. This variable is used in all of the playing piece arrays to keep track of individual sprites. The server will always have a value of 0 in the *who_am_i* variable. All other players will be assigned values by the server during connection.

What this means is that each player needs to know about him- or herself and the server. We don't need to worry about the other players in the game, so step 5 in our algorithm can be eliminated.

Code

The code for inputting the players based on our algorithm is

```
void CNetwar2View::OnPlayersInputplayers()
{
  CInputSelf Self_Dlg;
  CInputServer Server_Dlg;

  if (total_players == 0)
  {
  players[0] = new CPlayers;

  Self_Dlg.DoModal();
  total_players = atoi(Self_Dlg.m_total_players);

  //set gold in operations window
  ops_dlg->total_gold = 500000;
  ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);


  the_server = new CServer;
  if (Self_Dlg.type == 1)
{
     server = 0;
     who_am_i = 0;
}
else
{
     int ret = Server_Dlg.DoModal();

     strcpy(the_server->address, Server_Dlg.m_input_internet);
     server = 1;
```

```
      }
    }
  }
```

You will need to enter this code into the appropriate method in the netwavw.cpp file.

The code starts by checking the *total_players* variable. If the value is 0, then player information has not been entered yet. At this point we know that there is at least one player, so a Cplayer object is initialized. The IDD_INPUT_SELF dialog box is displayed to the user to gather information. Upon return from the dialog box, the variable *total_players* is set equal to the *m_total_players* member variable of the dialog box. At this point we have to do a little housekeeping and tell the operations window to display the total number of gold pieces for this player.

Once the gold is displayed, the code creates a server object. This server object performs a dual function. For the server machine, it keeps track of all of the socket information between the clients. For the clients, the server object will hold the Internet address of the server. When the player clicks on the server or client button on the IDD_INPUT_SELF dialog box, a member variable called *type* is set to 1 or 0 respectively. If the variable is a 1, then the code knows this player is the server. In this case, a variable called *server* is set equal to TRUE and the *who_am_i* variable is set to 0. If the player is not the server, a call is made to display the IDD_INPUT_SERVER dialog box. Upon return from this dialog box, the Internet (or telephone) number of the server is copied to a member variable of the server object. The variable *server* is set to a FALSE to indicate that this machine is not the server. That's all there is to inputting player information. What we know at this point is this:

- The object player[0] contains information about each player on each machine.
- A server object contains information about the server.
- The server has the variable *who_am_i* = 0 and the variable *server* = TRUE.
- All clients have the variable *server* = FALSE.

Before we go any further, we need to create the server object and look at any code that needs to be written for the two input dialog boxes.

Server Class

The purpose of the server class is to group information about the server into a convenient form. As we saw in Chapters 7 and 8, many different data structures are necessary for the socket connections. All of this information will be put into the server class. In addition, all of the clients will need to have the Internet address of the server, so this should be kept in the class as well.

Open a new file and place the following code in it:

```
#include "winsock.h"

class CServer
{
 public:
   int socket;
   struct sockaddr_in connection;

   BOOL active_connections[3];          // true if connection good
   int total_connected;                 // total players connected to the server
   int clients[3];                      // sockets for connected players
   struct sockaddr_in clients_addr[3];  // structures for connected players
   char address[25];                    // Internet address or telephone number

 public:
   CServer();
   ~CServer();
};
```

Save this file under the name server.h. This header file contains all of the information for the server class called CServer. The member variables are:

socket	initializes socket for server-to-client connections
connection	socket data structure for client socket initialization
active_connections	an array of Boolean values indicating whether or not a particular client's connection is valid
total_connected	a count of the total number of clients currently connected to the server
clients	the socket value for a connection to a particular client
clients_addr	socket data structures for each client connection
address	character array containing the Internet address or telephone number of the server—used *only* by clients

Now open a new file and put the following code in it.

```
#include "server.h"
#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
#include "netwavw.h"

CServer::CServer()
{
 int i;
 total_connected = 0;
 for(i=0;i<3;i++)
  active_connections[i] = FALSE;

}
CServer::~CServer()
{
}
```

Save this file as server.cpp. This is the code for the server class's constructor and destructor. The only code is found in the constructor, and it initializes the *active_connection* Boolean array to all FALSE values.

IDD_INPUT_SELF Class Code

You will remember that when a dialog box and associated class is created, additional code may be necessary for some of the actions of the dialog box. In the case of the IDD_INPUT_SELF dialog box, we need to create a member variable called *type* that is set to 1 for a server and 0 for a client.

Open the file inputsel.h and add the following code to the first PUBLIC area:

```
int type;
```

Now open the file inputsel.cpp and make the following changes to the OnRadioClient() and OnRadioServer() methods:

```
void CInputSelf::OnRadioClient()
{
 type = 0;
}

void CInputSelf::OnRadioServer()
{
 type = 1;
}
```

Each time one of the buttons is clicked, the *type* variable will be set to a value reflecting the player's choice.

IDD_INPUT_SERVER Class Code

In the IDD_INPUT_SERVER dialog box, the only thing the player is allowed to do is enter the Internet or telephone number of the server machine. No additional code is necessary in this dialog box.

Everything is ready for a compile. All of the new code can be found in the netwar2/chapt9/a directory. Since the server class was created manually, you will need to add it to the project file. Choose Project and Edit to add the server class CPP file.

In addition, you need to add the following include files to the top of the netwavw.cpp file

```
#include "inputsel.h"
#include "inputser.h"
```

and

```
#include "server.h"
```

to the top of the netwavw.h file. You will also need to add the declarations

```
int total_players;
int server;
CServer the_server;
```

to the netwavw.h file and the code

```
total_players = 0;
```

to the view constructor in netwavw.cpp.

Once you compile the application and run it, you can enter players. Click on Players and then Input Players. You should see the dialog box seen in Figure 9.6. Click on the Client button and enter the number 2 in the edit control for total players. This dialog box will disappear and the server dialog box will appear as in Figure 9.7. Enter the Internet address of a server and click OK.

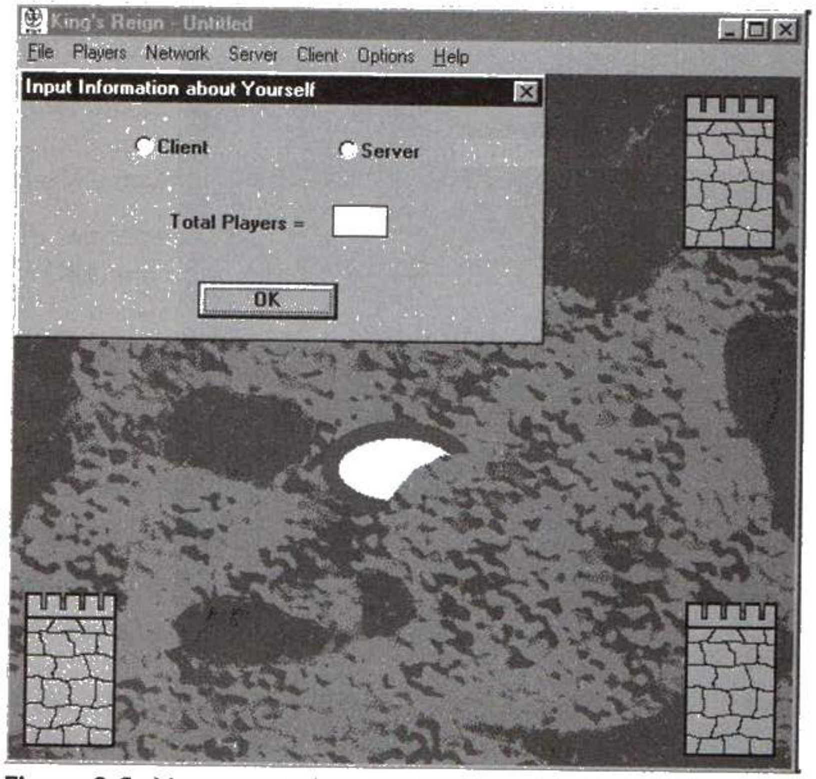

King's Reign - Untitled
File Players Network Server Client Options Help
Input Information about Yourself
Client
Server
Total Players =
OK

Figure 9.6 *Netwar2 windows application dialog box for your information.*

Buying Equipment

If you look back at the code for buying equipment, you will notice that we must have a value for the *who_am_i* variable in order to put sprites on the playing field. This means that if you don't input players and select yourself as a server, the variable *who_am_i* will have an unknown value and the program will abort. We need to make sure that the variable *who_am_i* has a legitimate value in it.

In the netwavw.cpp file, add the following code to the constructor.

```
who_am_i = -1;
```

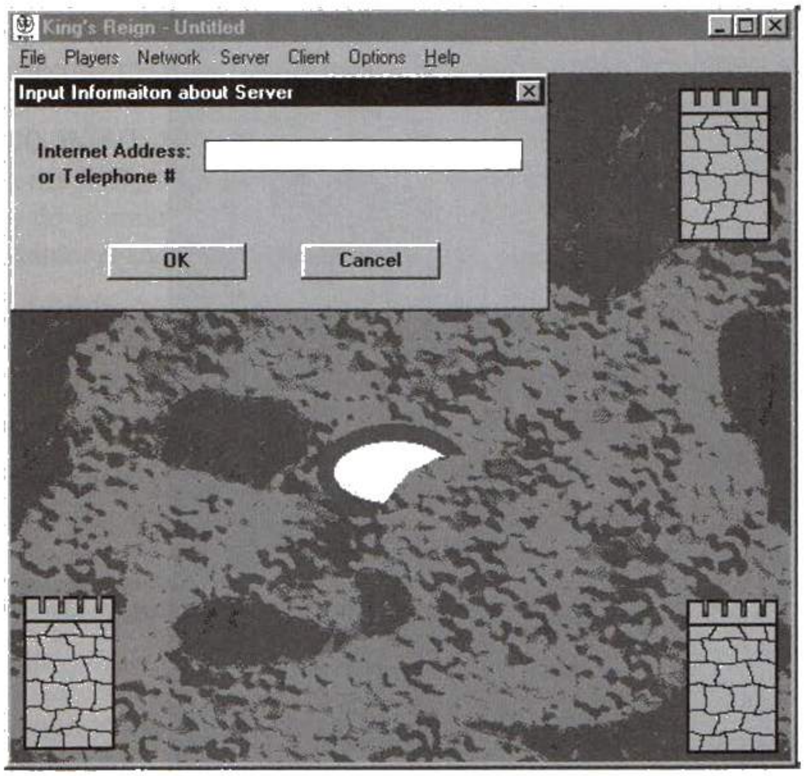

King's Reign - Untitled
File Players Network Server Client Options Help
Input Informaiton about Server
Internet Address:
or Telephone #
OK
Cancel

Figure 9.7 *Netwar2 windows application for your server's information.*

Now move to the buy_equipment() function and add the following IF statement around all of the code in the function.

```
if (who_am_i != -1)
{
 code here!!
}
else
 MessageBox("Enter Player Information First and Connect with Server", MB_OK);
```

If you now try to purchase equipment without first entering players and connecting to the server, you will receive an error message. There is a back door, however. If you enter players and set yourself as the server, you will be allowed to buy and place equipment on the playing field. You will only want to do this if you do not intend to have clients connect to you.

Initializing Winsock

After inputting the players in the game, we are ready to initialize Winsock. In setting up this code, we are going to prepare for using a modem and possibly the Windows for Workgroups network connections as well.

To initialize Winsock, the player would click on a menu item called NETWORK and then click on the Winsock - Internet entry. This will cause control to be transferred to a handler function and Winsock will be initialized; once we add the code necessary for the initialization.

The first step is to add the menu items. Start AppStudio and create a popup menu item with the caption "NETWORK". Under this menu item put the following entries

Winsock - Internet

Serial - Modem

Windows for WorkGroups

Now close the menu and bring up ClassWizard. Select the class CNetwar2View and create *both* command and update command handlers for the Object IDs.

ID_NETWORK_SERIAL_MODEM
ID_NETWORK_WINDOWSFORWORKGROUPS
ID_NETWORK_WINSOCKINTERNET

Save your change and close ClassWizard and AppStudio.

Using Update Command Handlers

Remember, when we added sound to our game, we used the update command handler for the Sound menu item to put a check mark beside the word Sound when sound was activated. We are going to do the same thing here to indicate which of the three different network connections is currently the active one.

The first step is to create a variable to hold the current network and three constants. In the netwavw.h file, enter the following constant declaration at the top of the file.

```
#define NETWORK_NONE            0
#define NETWORK_INTERNET        1
#define NETWORK_SERIAL          2
#define NETWORK_WORKGROUPS      3
```

Move to the class definition and add the following line in the first PRIVATE area.

```
int current_network;
```

Now open the netwavw.cpp file and add the following line to the constructor.

```
current_network = NETWORK_NONE;
```

These lines of code set up a variable called *current_network* that contains a value representing the current network. There are four possible values, 0 through 3, which are represented by constant strings. When the game is first executed, no network is established.

Now we will move to the functions that update the menu items. Find the following three methods and make the appropriate changes to them.

```
void CNetwar2View::OnUpdateNetworkSerialmodem(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_SERIAL);
}

void CNetwar2View::OnUpdateNetworkWindowsforworkgroups(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_WORKGROUPS);
}

void CNetwar2View::OnUpdateNetworkWinsockinternet(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_INTERNET);
}
```

Each time one of these menu items is clicked, the check mark will be placed if the variable *current_network* is equal to the appropriate network value.

The first step is to create a variable to hold the current network and three constants. In the netwavw.h file, enter the following constant declaration at the top of the file.

```
#define NETWORK_NONE            0
#define NETWORK_INTERNET        1
#define NETWORK_SERIAL          2
#define NETWORK_WORKGROUPS      3
```

Move to the class definition and add the following line in the first PRIVATE area.

```
int current_network;
```

Now open the netwavw.cpp file and add the following line to the constructor.

```
current_network = NETWORK_NONE;
```

These lines of code set up a variable called *current_network* that contains a value representing the current network. There are four possible values, 0 through 3, which are represented by constant strings. When the game is first executed, no network is established.

Now we will move to the functions that update the menu items. Find the following three methods and make the appropriate changes to them.

```
void CNetwar2View::OnUpdateNetworkSerialmodem(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_SERIAL);
}

void CNetwar2View::OnUpdateNetworkWindowsforworkgroups(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_WORKGROUPS);
}

void CNetwar2View::OnUpdateNetworkWinsockinternet(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_INTERNET);
}
```

Each time one of these menu items is clicked, the check mark will be placed if the variable *current_network* is equal to the appropriate network value.

Winsock Command Handler

Turning our attention to the command handler for Winsock, we find that the code for this handler simply needs to initialize the Winsock system, much as we did in the initialization functions of the test program in the previous chapter.

Find the OnNetworkWinsockinternet() method in the netwavw.cpp file and add the following code:

```
void CNetwar2View::OnNetworkWinsockinternet()
{
 WORD wVersionRequested;
 int err;

 //Start DLL find out if correct version
 wVersionRequested = 0x0101;
 err = WSAStartup(wVersionRequested, &wsaData);
 if (err != 0)
 {
  MessageBox("Winsock Version 1.1 not found...", "No Network", MB_OK | MB_ICONSTOP);
  return;
 }
 else
 {
  MessageBox("Winsock found...OK for use", "Found Network", MB_OK);
  current_network = NETWORK_INTERNET;
 }
}
```

The code starts out by calling the WSAStartup() function to initialize the system. If the return value is not equal to 0, then an error has occurred. In this case, a message box is displayed with an error statement. If the function returns successfully, a message box appears telling us that everything is installed and ready for use. It is in this code that the variable *current_network* is set equal to NETWORK_INTERNET.

One variable is used in this code that needs to be declared in our view class. This variable is *wsaData*. To declare this variable, add the following piece of code in the view class definition:

```
WSADATA wsaData;
```

As in the case of the other Winsock application we developed in Chapter 8, the file winsock.dll must be in your path or in the /windows directory on your hard drive in order for the game to run correctly. In addition, you will need to copy the file winsock.lib to the directory with the netwavw.* files and edit your project file to include this file.

Creating Connections

It's time to start making some connections. Since we have two different types of players, clients and a server, we will have two different ways of making all of the connections for the game. We will look at the server first.

Server Waits

When the server has finished entering all of the player data, it will know exactly how many players will be in the game. Out of this, the server will be able to determine how many connections or clients will be connecting to it. Using this number, the server can wait for all of the necessary connections before it allows the game to be played.

Before we start with the code, let's look at a general strategy. In the previous chapter, we saw that the Winsock system allows many connections to many computers at the same time. In the Windows environment, Winsock sends a message to our application when one of these connections is made. What we want to do is create a function that will make the preliminary connections and bindings with Winsock and associated Internet services. Once all of this has taken place, the code will let Windows know that we are waiting for connections and give Windows a function to call when a connection comes through.

Once this function is called, we will record in the server object that the connection that has been made, let the client at the other end of the connection know which player it is (the *who_am_i* value), and check to see if we have all of the clients attached.

In addition to all of this, we have to have several defensive mechanisms so that the server does not start to look for connections when this machine is not the server, or our current network is not Internet-based, or the players have not be entered.

The first thing we need to do is give our server player a way of telling the system it should start to listen for socket connections. The easiest way is to create a menu option. Start AppStudio and add a menu item called Wait For Connect. This menu option will not be a popup, so be sure to click the POPUP square to remove the check mark. Go into ClassWizard and create a handler for the menu item as well.

Now open up the netwavw.cpp file, find the added handler called OnWaitingForConnect and add the following code:

```
int len_connection_addr;

 PLM// This is for the server ONLY!!!
 if (server != 0)
 {
  MessageBox ("Sorry but you are not the server!!", "Not Server", MB_OK);
  return;
 }

 if (current_network== NETWORK_NONE)
 {
  MessageBox("You must select a Network Type first", "Error", MB_OK);
  return;
 }

 if (total_players < 2)
 {
  MessageBox("You must select your players", "Error", MB_OK);
  return;
 }
 if (current_network == NETWORK_INTERNET)
 {
  the_server->socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (the_server->socket < 0)
 {
  MessageBox("Unable to create socket", "No Socket", MB_OK);
  return;
 }

 len_connection_addr = sizeof(struct sockaddr_in);
 the_server->connection.sin_family = AF_INET;
 the_server->connection.sin_addr.s_addr = htonl(INADDR_ANY);
 the_server->connection.sin_port = htons(5002);
 if (bind(the_server->socket,(struct sockaddr *)&the_server->connection,
len_connection_addr) < 0)
 {
  MessageBox("Unable to bind socket", "No Bind", MB_OK);
  return;
 }

  listen(the_server->socket, 3);

  //register with ASYNC
  WSAAsyncSelect(the_server->socket, m_hWnd, MSG_ACCEPT, FD_ACCEPT);
 }
```

The code starts by making a check of the *server* variable. If this machine is the server machine, then this variable will have a value of 0. If this is not the case, a message box is displayed letting the user know the situation. The second security test comes next when the code checks to make sure that the variable *current_network* is equal to NETWORK_NONE. Again, if this is the case, a message box is displayed letting the user know. The last test is a check of the *total_players* variable. This variable needs to be greater than 1 in order for there to be a connection with another machine.

The remaining lines of code should look familiar, as they are the same used in the example programs in the previous chapter. Of particular importance is the last line of code. This line of code lets the Windows system know that we are waiting for connections to come over the socket *the_server->socket*. When a connection is detected, the message MSG_ACCEPT should be sent to the window application. The last parameter is a flag indicating that we are only interested in ACCEPT connections.

For the code to work correctly, we need to put in a message macro at the top of the code. In the message map, add the code

```
ON_MESSAGE(MSG_ACCEPT, OnMsgAccept)
```

This will cause the function OnMsgAccept() to be called when a connection comes through.

```
declare MSG_ACCEPT
declare OnMsgAccept function
```

Accepted Connections

Our server is now ready to accept connections. Somewhere around the world, a client will try to make a connection. When this happens, the Winsock code will post a message to our game's window letting us know a connection is being tried.

At this point, we have to get ourselves ready to complete the connection. We have seen that we ACCEPT the connection and create a socket dedicated to this client. The server can post to this socket and it will only communicate with this client. For the game, we have to do this much and more in order for this to work correctly.

First, we should get the code into the system. In the netwavw.cpp file, add the following handler.

```
long CNetwar2View::OnMsgAccept(UINT wParam, LONG lParam)
{
 int len, our_value;
 char buffer[70];

 our_value = the_server->total_connected;
 the_server->total_connected++;

 len = sizeof(struct sockaddr_in);
 the_server->clients[our_value] = accept(the_server->socket, (struct sockaddr
*)&the_server->clients_addr[our_value], &len);
 if (the_server->clients[our_value] < 0)
 {
  MessageBox("Error in Accept with client", "Error", MB_OK);
  return 0;
 }

 if (our_value == 0)
  WSAAsyncSelect(the_server->clients[our_value], m_hWnd, MSG_FROM_PLAYER1, FD_READ);
 else if (our_value == 1)
  WSAAsyncSelect(the_server->clients[our_value], m_hWnd, MSG_FROM_PLAYER2, FD_READ);
 else if (our_value == 2)
  WSAAsyncSelect(the_server->clients[our_value], m_hWnd, MSG_FROM_PLAYER3, FD_READ);

 the_server->active_connections[our_value] = TRUE;

 sprintf(buffer, "%11s %3d %3d %3d %3d %32s", "who", our_value+1, 0, 0, 0, "");
 send(the_server->clients[our_value], buffer, 60, 0);

 if (the_server->total_connected == total_players-1)
 {
  WSAAsyncSelect(the_server->socket, m_hWnd, 0, 0);

  SendInfo("game ready", 0, 0, 0, "");
  MessageBox("All Connected. Begin Game", "Go!", MB_OK);
 }

 return 1;
}
```

For this code to work correctly, we have to remember that we have registered with Winsock that this function needs to be called when a client tries to connect with the server. As we go through this code, you will find several references to routines and techniques that we have not yet discussed. We will talk about these new things as we see them.

The only time this function is called is when a client is trying to make a connection with us, the server. The server object has a member variable called *total_connected* that is a running count of the number of clients currently connected with the server. We will assume that no clients are currently connected, so *total_connected* is equal to zero. The value in *total_connected* is copied to a local variable called *our_value*. This variable will be used to initialize a place in the server arrays for the current client. After the value is copied into *our_value*, the variable *total_connected* is incremented to show that another client has connected. In our example *total_connected* is now equal to 1.

The next lines of code

```
len = sizeof(struct sockaddr_in);
 the_server->clients[our_value] = accept(the_server->socket, (struct sockaddr
*)&the_server->clients_addr[our_value], &len);
 if (the_server->clients[our_value] < 0)
 {
  MessageBox("Error in Accept with client", "Error", MB_OK);
  return 0;
 }
```

create a socket for the connection between the client and the server. The function that makes the connection is accept(). The socket for the client is located in the *the_server->clients[our_value]* member array. Since *our_value* is currently 0, we are using the first location in the array for this first client. To be safe, the code makes a quick check of the value in this array location to make sure that it is a valid socket identifier.

The next set of IF statements does one of the most important jobs of this code, which is to assign a message and associated handler for communication with this client. If you look at the preceding code, you will see that we are telling Winsock that when it performs a read (last parameter) on the first client's socket (first parameter) it should send the message MSG_FROM_PLAYER1 (second parameter) to our application.

In the message map for our game, we have the entry

```
ON_MESSAGE(MSG_FROM_PLAYER1, OnMsgFromPlayer1)
```

which maps the message to an appropriate message handler. At this point, everything has been successfully set up between the client and the server. Each of the machines can communicate freely with the other. Before we end, the code indicates that the current client connection is an active connection by setting the appropriate array position to TRUE.

Who Am I? In our algorithm for the client/server connection, we mentioned that each of the clients would need to be given their *who_am_i* value from the server. The next two lines of code does the job for us.

```
sprintf(buffer, "%11s %3d %3d %3d %3d %32s", "who", our_value+1, 0, 0, 0, "");
 send(the_server->clients[our_value], buffer, 60, 0);
```

What we are doing here is setting up a packet of length 60 that contains six different fields. This packet structure will be used throughout the entire game for all communications between the server and clients. The first field of the packet is the command that the client should execute. In this case the command is called "who". We will see a little later that each client has a receive from server function that takes the packet and sends it to a parser that performs the necessary command. This function will be documented in the Client Tries section. The second through fifth fields in the packet are available for integer values. The last field is a string field.

The WHO command uses the first integer field to send the *who_am_i* value to the client. This value is always one value higher than the value at the server, since the server always has a value of 0 for its *who_am_i* variable

End of Function The last section of code in this function checks to see if the total number of connections is appropriate for the number of total players in the game. If this is the case, all of the clients have connected and several tasks must be executed.

```
if (the_server->total_connected == total_players-1)
  {
   WSAAsyncSelect(the_server->socket, m_hWnd, 0, 0);

   SendInfo("game_ready", 0, 0, 0, "");

   MessageBox("All Connected. Begin Game", "Go!", MB_OK);
}
```

Since there are not going to be any more connections with the server, we can tell Winsock to stop telling us about other machines trying to connect with us. At this point, all of the clients are connected and the game is ready to be played. Using a function called SendInfo(), the server sends the command to all of the clients called GAME_READY. This command indicates to the player's code that the game is ready to be played. To indicate to the server player that the game is ready for play, a message box is displayed with a short message indicating the game start situation. The message box appears as shown in Figure 9.8.

SendInfo() Function One of the new functions in the preceding code is called SendInfo(). The purpose of this function is to send a specific command to all of the clients connected to the server. This function really isn't special, since we could just create a packet and immediately send it to all of the clients just as we sent the command WHO to clients as they connected.

The reason for this new function is twofold. The first is to keep from putting many lines of code in the game each time we need to send something to the clients. The second reason is related to the first in that we will be expanding the game to include modem connections and this will add considerable code to the send routines.

You need to add the following function prototype to our view class in netwavw.h.

```
void SendInfo(char *command, int data1, int data2, int data3, char *msg)
```

A member variable is necessary as well. Add the line

```
char send_buffer[70];
```

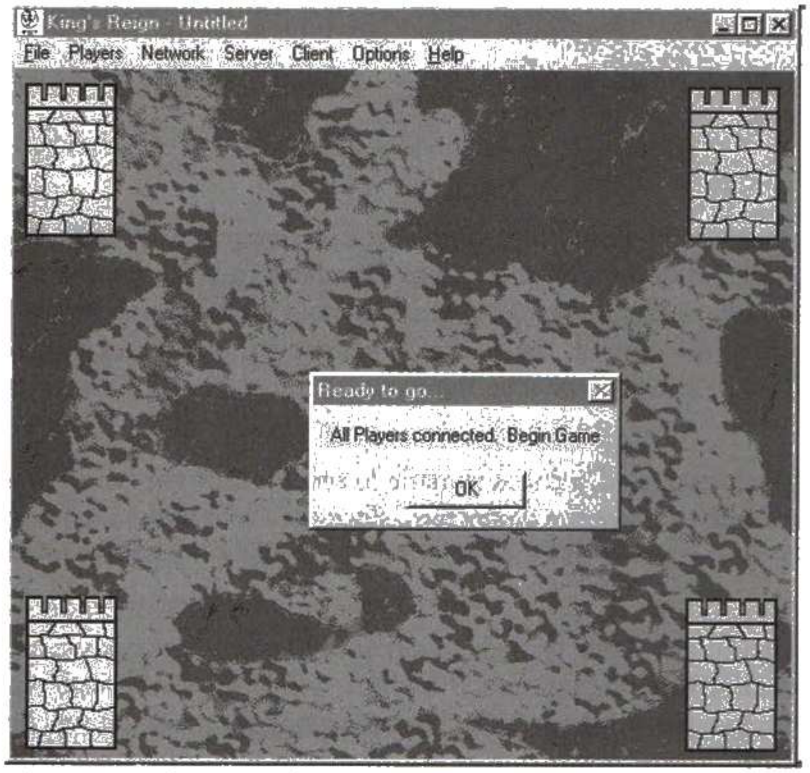

King's Reign - Untitled
File Players Network Server Client Options Help
Ready to go...
All Players connected. Begin Game
OK

Figure 9.8 *The Go! dialog box.*

to our view object class. Now add the function to the .cpp file.

```
void CNetwar2View::SendInfo(char *command, int data1, int data2, int data3, char
*msg)
{
 int i;

 sprintf(send_buffer, "%11s %3d %3d %3d %3d %32s", command, data1, data2, data3,
who_am_i, msg);

 if(server == 0)
 {
  if (current_network== NETWORK_INTERNET)
 {
  for (i=1;i<total_players;i++)
     send(the_server->clients[i-1], send_buffer, 60, 0);
 }
}
else
{
   if (current_network== NETWORK_INTERNET)
      send(players[0]->socket, send_buffer, 60, 0);
  }
}
```

There are two parts to the SendInfo() command. The first part deals with the server and the second deals with the clients. Common to both parts is the packet creation code. Using a *sprintf()* statement and a global variable called *send_buffer*, each of the parts of the packet is assembled. Notice the fourth integer field. This field will always be the value of the player who has sent the packet.

Once the packet is assembled, a check is made to see if we are the server or a client. It is important to make the distinction, since the client will be sending the packet only to the server and the server will be sending the packet to *all* clients.

If the sender is a client, a check is made to determine which network we are using. Since we only have the Internet, this check will be true. Although we haven't seen this yet, the client/server socket connection for the client will be stored in the *socket* member variable of the client's *player[0]* variable. The code uses a Winsock send() command to send the packet to the server. The server, on the other hand, does a quick loop of all of the clients currently connected and sends the packet to each of them.

Client Message Handlers Once a client attaches to the server, the server lets Winsock know that it needs to send a message when any of the clients sends a packet to the server. As we saw, the messages are

MSG_FROM_PLAYER1
MSG_FROM_PLAYER2
MSG_FROM_PLAYER3

You need to add these messages to the message map at the top of the netwavw.cpp file.

```
ON_MESSAGE(MSG_FROM_PLAYER1, OnMsgFromPlayer1)
```

```
     ON_MESSAGE(MSG_FROM_PLAYER2, OnMsgFromPlayer2)
     ON_MESSAGE(MSG_FROM_PLAYER3, OnMsgFromPlayer3)
```

Now the handlers need to be added.

```
long CNetwar2View::OnMsgFromPlayer1(UINT wParam, LONG lParam)
{
 int i;
 char buffer[70];

 i = recv(the_server->clients[0], buffer, 60, 0);
 buffer[i] = '\0';

 SendToOthers(0, buffer);

 DoCommand(buffer);
 return 1;
}

long CNetwar2View::OnMsgFromPlayer2(UINT wParam, LONG lParam)
{
 int i;
 char buffer[70];

 i = recv(the_server->clients[1], buffer, 60, 0);
 buffer[i] = '\0';

 SendToOthers(1, buffer);

 DoCommand(buffer);
 return 1;
}


long CNetwar2View::OnMsgFromPlayer3(UINT wParam, LONG lParam)
{
 int i;
 char buffer[70];

 i = recv(the_server->clients[2], buffer, 60, 0);
 buffer[i] = '\0';

 SendToOthers(2, buffer);

 DoCommand(buffer);
 return 1;
}
```

In addition, you will need to add prototypes for each of these functions in the view class. The code is the same in each of the message handlers except for one thing: The index value in the server variables for the socket connections with each client. In other words, when Winsock detects that player 1 is sending something to the server, it will send the message MSG_FROM_PLAYER1 to our game. Our game will process the message and transfer control to the function OnMsgFromPlayer1(). This function will call the function rec() with the socket in array position 0. Once the packet from player 1 is received, it will put an end-of-line character on the end and proceed to send the packet to other players and then process the packet.

If player 2 were sending the packet, the code would have to look at array position 1 instead of 0. The same is true for player 3, who uses array position 2. Now, in the process of receiving something from player 1, it was mentioned that the packet was forwarded to the other players. When player 1 sends a packet to the server, he or she is actually sending the packet to all of the players in the game, not just the server. It is the responsibility of the server to forward the packet to all other players in the game. This is true no matter which player sends a packet to the server. This forwarding is done through the function SendToOthers().

SendToOthers() The function SendToOthers() is where the server forwards a packet from one player to all players in the game. The code is:

```
long CNetwar2View::SendToOthers(int client, char *buffer)
{
 int i, length;

 length = strlen(buffer);
 switch(client)
 {
   case 0: if (the_server->total_connected > 1) i = send(the_server->clients[1],
buffer, 60, 0);
       if (the_server->total_connected > 2) i = send(the_server->clients[2],
buffer, 60, 0);
       break;

   case 1: i = send(the_server->clients[0], buffer, 60, 0);
       if (the_server->total_connected > 2) i = send(the_server->clients[2],
buffer, 60, 0);
       break;

   case 2: i = send(the_server->clients[0], buffer, 60, 0);
       i = send(the_server->clients[1], buffer, 60, 0);
       break;
```

```
  }
  return 1;
}
```

Add this code as well as a prototype to your code. The function is just one big *switch*. Based on the player who sent the packet, the server determines which of the players to forward the packet to. Now, the server cannot just forward the packet to all players, since they might not be connected. Let's take the situation where player 1 has just sent a packet to the server and only two players are playing the game. The *switch* statement will use the first case, *case 0,* since player 1 is a zero in the server's eye. The code will check the *total_connected* variable to see if other players are connected. Since there are only two players in the game, this variable will have a value of 1. Each of the IF statements in the *case 0* code will evaluate to FALSE and the packet will not be forwarded. If there are three players in the game, the packet will be sent to player 2 only and not player 3.

OnCommand()

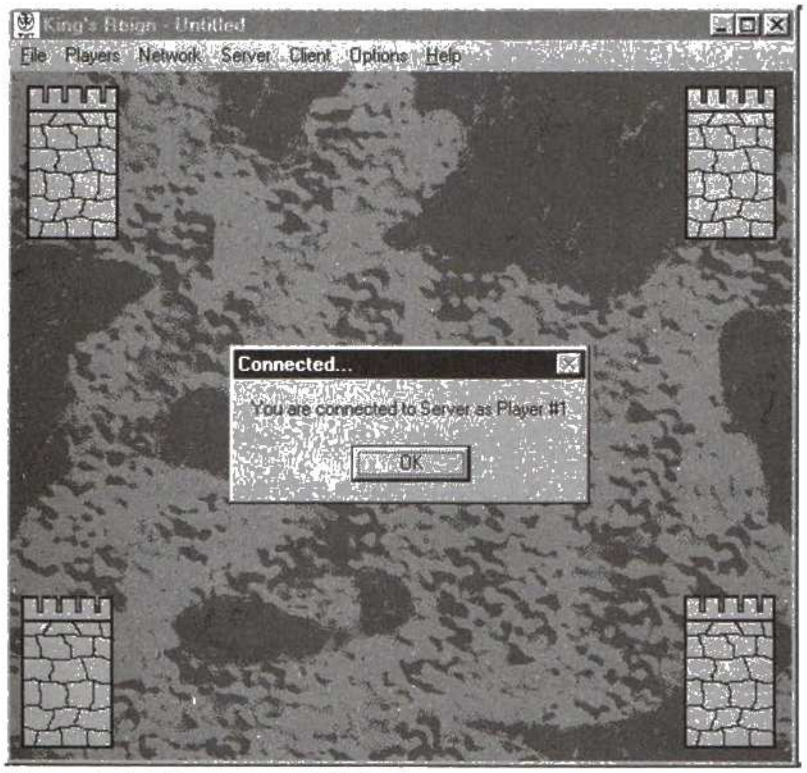
The last thing you will see in each of the player message handlers is a call to the function OnCommand(). This function is where the commands from the players are parsed and executed. In our final code this function is rather large, but here is the beginning. Add the code to the source file and the prototype to the header of our view class.

```
long CNetwar2View::DoCommand(char *buffer)
{
 int player1, i, j;
 int player2;
 int sprite1;
 int sprite2;
 char command[60], msg[45];

 sscanf(buffer, "%s %d %d %d %d %s", command, &player2, &sprite1, &sprite2,
&player1, msg);

 if (strcmp(command, "who") == 0)
 {
  if (current_network!= NETWORK_SERIAL)
  {
   sprintf (command, "You are connected to Server as Player #%d", player2);
   MessageBox(command, "Connected...", MB_OK);
  }
```

```
    who_am_i = player2;
  }
return 1;
}
```

The DoCommand() function is fairly straightforward. The packet is decoded into separate parts using a *sscanf* statement. The command part of the packet is string compared to static word such as "who" in the preceding function.

You will recall that the WHO packet used the first integer field for the *who_am_i* value. When a client receives this packet, it will set its *who_am_i* variable to this value and display a message indicating to the player which number the player is. The message box that is displayed when player 1 attaches to the server is shown in Figure 9.9.



You are connected to Server as Player #1

Figure 9.9 *The connected dialog box.*

Client Tries

All of the code above is necessary just for the server to wait for connections from the clients. Parts of the code are not directly necessary but do show up, so they need to be discussed. Now we will move to the clients and the code they need to connect with the server.

The first step is to create a menu item, just as we did for the server. Open AppStudio and create a menu item called Connect To Server. This menu item is *not* a popup, so be sure to remove the check mark in the Popup box. Open ClassWizard and create a command handler for the menu item. Close Class-Wizard and AppStudio.

Now add the following code to the OnConnecttoserver() function in net-wavw.cpp.

```
void CNetwar2View::OnConnecttoserver()
{
    int len_connection_addr;

    if (current_network == NETWORK_NONE)
    {
     MessageBox("You must select a Network Type first", "Error", MB_OK);
     return;
    }

    if (total_players < 2)
    {
     MessageBox("You must select your players", "Error", MB_OK);
     return;
    }

    if (server == 0)
    {
     MessageBox("You are the Server - Use Waiting For Connect..", "Error", MB_OK);
     return;
    }

    if (current_network == NETWORK_INTERNET)
    {
     if (total_players > 4)
     {
      MessageBox("Sorry but you can only have 4 players on a network connection",
"Error", MB_OK | MB_ICONSTOP);
     }
     else
```

```
      {
        len_connection_addr = sizeof(struct sockaddr_in);
        players[0]->connection.sin_family = AF_INET;
     players[0]->connection.sin_addr.s_addr = inet_addr(the_server->address);
       players[0]->connection.sin_port = htons(5002); //the players[server]-
>get_port());

       players[0]->socket = socket(AF_INET, SOCK_STREAM, 0);
       if (players[0]->socket < 0)
       {
        MessageBox("Unable to create socket", "No Socket", MB_OK);
        return;
       }

       if (connect(players[0]->socket, (struct sockaddr *)&players[0]->connection,
len_connection_addr) < 0)
       {
        MessageBox("Unable to create a connection to server", "No Connection",
MB_OK);
        return;
       }

       //register with ASYNC
       WSAAsyncSelect(players[0]->socket, m_hWnd, SERVER_MSG, FD_READ);
     }
   }
}
```

You will need to add the following message definition to the netwar2.h file.

```
#define SERVER_MSG WM_USER + 17
```

Now let's take a look at the code. It starts out much like the waiting for connect function for the server. The code checks to make sure that a network has been selected, it checks to make sure that we are a client, and it checks for more than one player.

After all of the checks, it enters a body of code if the current network is Winsock. After that, there is code that should look familiar, as it was used in Chapter 8. This code sets up the client to make the connection to the server. The next-to-last operation of the code is to call the Winsock function connect() to try a connection to the server. If the connection is successful, the client lets Winsock know that all packets from the server should be met with a message SERVER_MSG being sent to the application.

In the message map, you need to add the code

```
ON_COMMAND(SERVER_MSG, OnServerSend)
```

which will tie our message about a packet from the server with the function OnServerSend(), which will actually receive the packet.

Socket to Players

Before we look at the code for receiving information from the server, we need to make three changes to the CPlayer class. Open the file player.h and make the code match the following:

```
#include "winsock.h"


class CPlayer
{
public:
    long gold_coins;
    int kills;
    int socket;
    struct sockaddr_in connection;

public:
     CPlayer();
     ~CPlayer();
};
```

What we have done is add a variable for the socket connection between the client and the server.

Receive from Server

When a packet is sent from the server to a client, the message SERVER_MSG is sent to our game. The game will process the message and execute the function OnServerSend() to process the incoming information.

Add the following function to the source file and create an appropriate prototype in the header file of our view class.

```
long CNetwar2View::OnServerSend(UINT wParam, LONG lParam)
{
 char buffer[70];
 int length;

 length = recv(players[0]->socket, buffer, 60, 0);
 buffer[length] = '\0';
```

```
  DoCommand(buffer);
  return 1;
}
```

The code simply reads the packet from the socket that has been created between the client and the server. The packet is appended with an end-of-line character and sent to the DoCommand() function for processing.

Disconnect

Once the game has been finished, the players need to disconnect the socket connections. The easiest way to do this is to allow the view object to handle it. Add the following code to the destructor in the netwavw.cpp file:

```
int i;
    if (current_network == NETWORK_INTERNET)
    {
     for (i=0;i<total_players;i++)
     {
    if (server == 0)
    {
     if (the_server->active_connections[i] == TRUE)
       closesocket(the_server->clients[i]);
    }
    else
     closesocket(players[0]->socket);

    delete players[0];
  }
 while (WSACleanup() != 0);
 }
```

The code closes all of the open and active connections and removes the Winsock library from memory. This is the same code used in the previous chapter.

Compiling and Running

All of the code up to this point can be found in the /chapt9/b directory. To see the code in action, execute the file netwar2.exe on a computer connected to the Internet. Record the IP address of this machine.

Now select the Players menu item and Input Players. Enter 2 players and click on Server. Click on OK. Select the network to use by clicking on Network and then Winsock. You should receive a message telling you that Winsock is OK to use. If you receive an error message, then you will need to install Winsock on your machine. You are ready to receive clients. Click on Wait For Connect.

Move to another machine connected to the Internet. Run netwar2.exe. Select Players and Input Players. Enter 2 players and click on Client. Now click on OK. The server dialog box will appear. Enter the IP address of the server machine in the edit line. Click on OK. Click on Network and then Winsock. Now click on Connect To Server. You will see a dialog box appear on the screen telling you that you are player #1.

Everything is working just as it should. You are not limited to only one client. You can run the game again and select four players in the game. In fact, with a little additional code, you could support many more than four players in the game. The remainder of this chapter will deal with converting all of the functionalities of the game to a multiplayer format.

Transferring Equipment Purchases

In normal game play, the next thing to happen after the players are connected is that the players would purchase equipment. All of the equipment that a player purchases must be made known to all other players. This does not mean that each player will know what all other players have in the way of equipment. Only the underlying code of the game will know this information. It must have this information so that the correct playing pieces can be displayed on the playing field. Remember that we will see our opponent's playing pieces on our playing field.

What we want to do here, is every time a player goes into the Buy Equipment dialog box and makes a purchase, the number and type of equipment purchased has to be sent all players. If you recall the Buy Equipment function, you will remember that there is an IF statement for each of the four playing pieces. If the member variable for the playing pieces is greater than 0, then we know some of these pieces have been purchased. This might be a good place to let the other players know about the purchases.

In order for the other players to set things up correctly on their end, what are they going to need to know about the playing pieces? They will need to know which player purchased the playing pieces, the playing piece type(s), and the number of playing pieces purchased. All of this information will need to be kept by all of the players, and a sprite object will need to be created for all sprites the players create.

You will recall that we have a variable called *player_pieces*. This variable is a two-dimensional array of sprite objects. The first dimension is the player; the second is the sprite. Since we are creating sprites on other machines that represent sprites on our own machine, we will have to make sure that a one-to-one relationship is kept between the two arrays. In other words, if a legion playing piece is in location *playing_pieces[1][0]* and an archer in *playing_pieces[1][1]*, then all players must have the same information in their arrays.

So how are we going to send all of this information to the other players? The answer is a packet and the SendInfo() command. In your Buy_Equipment() function, you will see code like this

```
if (pDlg.cavalrys_bought > 0)
  {
    for(i=0;i<pDlg.cavalrys_bought;i++)
    {
    playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
    if (playing_pieces[who_am_i][total_pieces[who_am_i]]->Initialize(GetDC(),
bitmaps[CAVALRY_BITMAP_MASK], bitmaps[CAVALRY_BITMAP],
                              bitmaps[CAVALRY_BITMAP_HIGHLIGHT],
                     FALSE, 4, 3, who_am_i, "Cavalry", 40) == FALSE)
     MessageBox("Unable to initialize dragon sprite", "Error", MB_OK);

    strcpy(ops_dlg->entries[ops_dlg->total_armies], "Cavalry");
    ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
    ops_dlg->total_armies++;

    total_pieces[who_am_i]++;
   }
   }
```

Add the following code between the last two }'s in the code above:

```
SendInfo("create_sprt", 1, pDlg.cavalrys_bought, 0, "");
```

So what are we doing? The code checks to see if we purchased any cavalry playing pieces. If we did, the code enters a loop, creates the necessary sprite objects on our machine, and then registers the objects with the armies list box on the operations window. This is the standard buy routine. What we have done is add a SendInfo command after the loop. The SendInfo command is sending a packet with the command CREATE_SPRT as the indicator to other players that they should create some sprites. The sprite that they should create is listed in the second parameter. The value of 1 indicates that cavalry sprites should be created. The third parameter indicates the number of cavalry sprites to create. The fourth and fifth parameters are not used.

By putting the SendInfo command in this location, we are able to guarantee that the one-to-one relationship will hold up between sprites. The packet will be sent to each of the players in the game. When the packet is received, it will be sent to the OnCommand() function to be processed.

Add the following code to the OnCommand() function just after the first IF statement:

```
else if (strcmp(command, "create_sprt") == 0)
  {

    for(i=0;i<sprite1;i++)
    {
     playing_pieces[player1][total pieces[player1]] = new CSprite;
     switch(player2)
     {
      case 1: if (playing pieces[player1][total pieces[player1]]-
>Initialize(GetDC(), bitmaps[player1][FLYING_BITMAP_M],
bitmaps[player1][CAVALRY_BITMAP],
                             bitmaps[player1][CAVALRY_BITMAP_H],
bitmaps[player1][FLYING_BITMAP_M],
                        FALSE, 4, 3, player1, "Cavalry", 4) == FALSE)
              MessageBox("Unable to initialize CAVALRY sprite", "Error", MB_OK);
           break;
    }
    total_pieces[player1]++;
   }
}
```

The code will check to see if the command of the packet is CREATE_SPRT. If it is, the code enters a loop based on the *sprite1* local variable. This variable relates to the second integer field of the packet, which just happens to be the count value for the playing pieces.

The creates a new sprite object in the next available position of the *playing_pieces* array. A *switch* statement is used to select the appropriate sprite to create, and it is created. Finally, the *total_pieces* variable is incremented to indicate the addition of this sprite.

This sequence occurs for the total number of playing pieces this player purchased of that particular type. All we need to do now is add three more SendInfo commands to the Buy Equipment code and add the necessary *case* statements to our OnCommand code.

The necessary SendInfo commands are

for the flyers: SendInfo("create_sprt", 4, pDlg.flyers_bought, 0, "");

for the archers: SendInfo("create_sprt", 2, pDlg.archers_bought, 0, "");

for the legions: SendInfo("create_sprt", 3, pDlg.legions_bought, 0, "");

The statements should all be placed just after the loop for each of the different playing pieces. We also need the *case* statements in the OnCommand() function. The necessary code is

```
case 2: if (playing_pieces[player1][total_pieces[player1]]->Initialize(GetDC(),
    bitmaps[player1][FLYING_BITMAP_M], bitmaps[player1][ARCHER_BITMAP],
       bitmaps[player1][ARCHER_BITMAP_H], FALSE, 4, 1, player1, "Archer", 3)
== FALSE)
        MessageBox("Unable to initialize archer sprite", "Error", MB_OK);
       break;

    case 3: if (playing_pieces[player1][total_pieces[player1]]->Initialize(GetDC(),
    bitmaps[player1][FLYING_BITMAP_M],      bitmaps[player1][LEGION_BITMAP],
       bitmaps[player1][LEGION_BITMAP_H], FALSE, 8, 6, player1, "Legions", 2)
== FALSE)
        MessageBox("Unable to initialize legion sprite", "Error", MB_OK);
       break;

    case 4: if (playing_pieces[player1][total_pieces[player1]]->Initialize(GetDC(),
    bitmaps[player1][FLYING_BITMAP_M], bitmaps[player1][FLYING_BITMAP],
       bitmaps[player1][FLYING_BITMAP_H], FALSE, 3, 3, player1, "Flyers", 9)
== FALSE)
        MessageBox("Unable to initialize flyer sprite", "Error", MB_OK);
        break;
```

Before we finish this section, we have to add one piece of housekeeping code to our view's constructor. The code is

```
for (i=0;i<4;i++)
 total_pieces[i] = 0;
```

This code will initialize all of the playing piece counts to zero. Once you have entered all of the code, compile it to make sure that you have copied everything correctly.

Additional Playing Pieces

If you read the code well, you will see that the *bitmaps* array has been extended from one dimension to two. The reason for this is each player will have its own set of playing pieces with some kind of color border used to distinguish them.

Therefore, we need to create 36 more bitmaps using AppStudio. Each of the playing pieces should look the same except the design should be outlined in some color. You could have red for player 0, blue for player 1, green for player 2, and purple for player 3. The bitmaps will have to be distinguished by their names as well.

So player 0's bitmaps might be called

IDB_P0_CAVALRY_BITMAP
IDB_P0_CAVALRY_MASK
IDB_P0_CAVALRY_HL

IDB_P0_LEGION_BITMAP
IDB_P0_LEGION_MASK
IDB_P0_LEGION_HL

IDB_P0_ARCHER_BITMAP
IDB_P0_ARCHER_MASK
IDB_P0_ARCHER_HL

IDB_P0_FLYER_BITMAP
IDB_P0_FLYER_MASK
IDB_P0_FLYER_HL

After you have created all of the bitmaps using AppStudio, they will need to be read into the system. Copy all of the code for reading in the bitmaps in the constructor of our view object and duplicate it three times.

One change will need to be made to all of the code however. In the statements

```
bitmaps[LEGION_BITMAP] = new CBitmap;
 if (bitmaps[LEGION BITMAP]->LoadBitmap(IDB_LEGION_BITMAP) == FALSE)
  MessageBox("Unable to load legion bitmap", "error", MB_OK);

 bitmaps[LEGION_BITMAP_MASK] = new CBitmap;
 if (bitmaps[LEGION_BITMAP_MASK]->LoadBitmap(IDB_LEGION_BITMAP_MASK) == FALSE)
  MessageBox("Unable to load legion mask bitmap", "error", MB_OK);

 bitmaps[LEGION_BITMAP_HIGHLIGHT] = new CBitmap;
 if (bitmaps[LEGION_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_LEGION_BITMAP_HL) == FALSE)
  MessageBox("Unable to load legion highlight bitmap", "error", PLLMB_OK);
```

all of the references to the bitmaps array will need to have the player number as the reference for the first dimension. To read all of the legion bitmaps for player 0, you would code things as

```
 bitmaps[0][LEGION_BITMAP] = new CBitmap;
 if (bitmaps[0][LEGION BITMAP]->LoadBitmap(IDB_P0_LEGION_BITMAP) == FALSE)
  MessageBox("Unable to load player0's legion bitmap", "error", MB_OK);

 bitmaps[0][LEGION_BITMAP_MASK] = new CBitmap;
 if (bitmaps[0][LEGION_BITMAP_MASK]->
         LoadBitmap(IDB_P0_LEGION MASK) == FALSE)
  MessageBox("Unable to load player0's legion mask bitmap", "error", MB_OK);

 bitmaps[0][LEGION BITMAP_HIGHLIGHT] = new CBitmap;
 if (bitmaps[0][LEGION_BITMAP_HIGHLIGHT]->
         LoadBitmap(IDB P0 LEGION_HL) == FALSE)
  MessageBox("Unable to load player0's legion highlight bitmap", "error", MB_OK);
```

You will also have to change the declaration of the *bitmaps* array from

```
CBitmap* bitmaps[48];
```

to

```
CBitmap* bitmaps[4][12];
```

The last change that you need to do is find all of the references to *bitmaps* in the Buy Equipment method and change it from

```
bitmaps
```

to

```
bitmaps[who_am_i]
```

This will reflect the additional bitmaps for each player. To see the effect of all of the additional bitmaps and the code for loading them, look at the net-wavw.cpp file in the directory /netwar2/chapt9/c.

Testing It

Since we have added quite of bit of processing and we need to make sure that everything works correctly, we should test our game to this point. The way that we can make a quick test is to put a message box in each of the case statements in the create_sprt command. This statement will tell us when a sprite is being created.

You can execute the program and have some number of players connect to the server. One of the players can purchase some playing pieces. All of the machines in the game should display a message box for each playing piece purchased. If this works, then everything is good to this point.

The code to add is

```
MessageBox("Creating a Cavalry Playing Piece", "Cavalry", MB_OK);
```

Put a message box statement like this inside of each case statement. Be sure to change the cavalry text to reflect the playing piece that is being created. Now compile and execute your code. All of the code can be found in the /netwar2/chapt9/c directory.

Registering New Sprites

With our code working correctly and all players able to recognize when a player purchases playing pieces, it's time to expand things. Once a player has purchased playing pieces, they will be displayed in their Armies list box in the operations window. The player will typically go to the list box and select an army to place on the playing field. When they do this, the "Armies" name is replaced and the playing piece appears on the playing field. Up to this point, the playing piece has appeared in the upper-left corner.

With four players, we should probably section the playing field into quarters and have each player's playing pieces appear on the field in one of the four corners. The breakdown will be as follows:

player 0 will appear in upper-left corner

player 1 will appear in lower-right corner

player 2 will appear in upper-right corner

player 3 will appear in lower-left corner

This places the players far enough apart to give them time to think about their moves. With the addition of opponents, placing the playing pieces in the corners is not our only concern. We also need to let each of the players know when we have double-clicked on an army. Our army has to appear on each of their screens as well. We will do this using another command that will be sent in a packet to all players. The command is called sprite_show and is used in the statement

```
SendInfo("sprite_show", ops dlg->selected_army, 0, 0, "");
```

This statement should be placed in the method

```
CNetwar2View::OnArmySelected(UINT wParam, LONG lParam)
```

located in the netwavw.cpp file. This is the method that is called when players double-click on one of their armies in the Armies list box of the operations window. The army that has been selected is in the member variable called *selected_army*. The SendInfo command sends a packet to the other players letting them know that we have selected an army and we give them the army number.

Our code will need to have a *strcmp* statement in the OnCommand() function to recognize our new command. The code looks like

```
else if (strcmp(command, "sprite_show") == 0)
  {
```

```
  playing_pieces[player1][player2]->active = TRUE;
  Invalidate(TRUE);
 }
```

This code is exactly the same as the code in the OnArmySelected() method except that the array values are sent from the player that actually selected the army. The value in player1 *always* relates to the player that sent the packet. The value in player2 is the value of the army selected. The playing piece that was selected by the player will be set active on *all* opponent machines. The Invalidate(TRUE) statement will cause the opponent's screens to be redrawn, and our playing piece will appear on the playing field.

When the Invalidate(TRUE) statement is actually executed, a message will be sent to our game telling it to redraw the screen. This will cause the OnDraw() method to be executed. Since this playing piece is being put on the playing field for the first time, we have to tell it which corner to place it in. Currently the code looks like this:

```
for (j=0;j<1;j++)
 {
  for(i=0;i<total_pieces[j];i++)
  {
   if (playing_pieces[j][i]->active == TRUE)
   {
    if (playing_pieces[j][i]->show)
    {
     playing_pieces[j][i]->Redraw(pDC);
    }
    else
    {
     playing_pieces[j][i]->Start(pDC, 50, 50);
     playing_pieces[j][i]->show = TRUE;
    }
   }
  }
 }
```

Notice that the first loop only looks at the playing pieces for player0. We need to change the code

```
j<1
```

to

```
j<total_players
```

The thing we are really concerned with is the code in the *else* statement. This code tells the system to start the new playing piece at location 50,50 no matter what. We need to make a little change here. Replace the code in the first set of {'s after the *else* statement to

```
switch (playing_pieces[j][i]->player)
          {
             case 0: playing_pieces[j][i]->Start(pDC, 50, 50);
                     break;

             case 1: playing_pieces[j][i]->Start(pDC, 400, 350);
                     break;

             case 2: playing_pieces[j][i]->Start(pDC, 50, 350);
                     break;

             case 3: playing_pieces[j][i]->Start(pDC, 400, 50);
                     break;
          }
          playing_pieces[j][i]->show = TRUE;
```

Now when a new playing piece is added to the playing field, a *switch* statement based on the owner of the playing piece is used to place the playing piece in one of the corners as in Figure 9.10.

Each time any players select an army from their armies list box, a packet will be sent to each of the players that contains the command *sprite_show,* the player's number, and the number of the playing piece to show. Once the packet is received, the command will be evaluated and playing piece's ACTIVEmember variable will be set to TRUE. An Invalidate(TRUE) statement will cause the playing piece to be drawn on the screen in a corner designated for that player.

You can find the code to this point in the directory /netwar2/chapt9/d.

Moving Sprites

Things are moving along well now. Once players have placed playing pieces on the screen, they are going to move the pieces eventually. We had better be

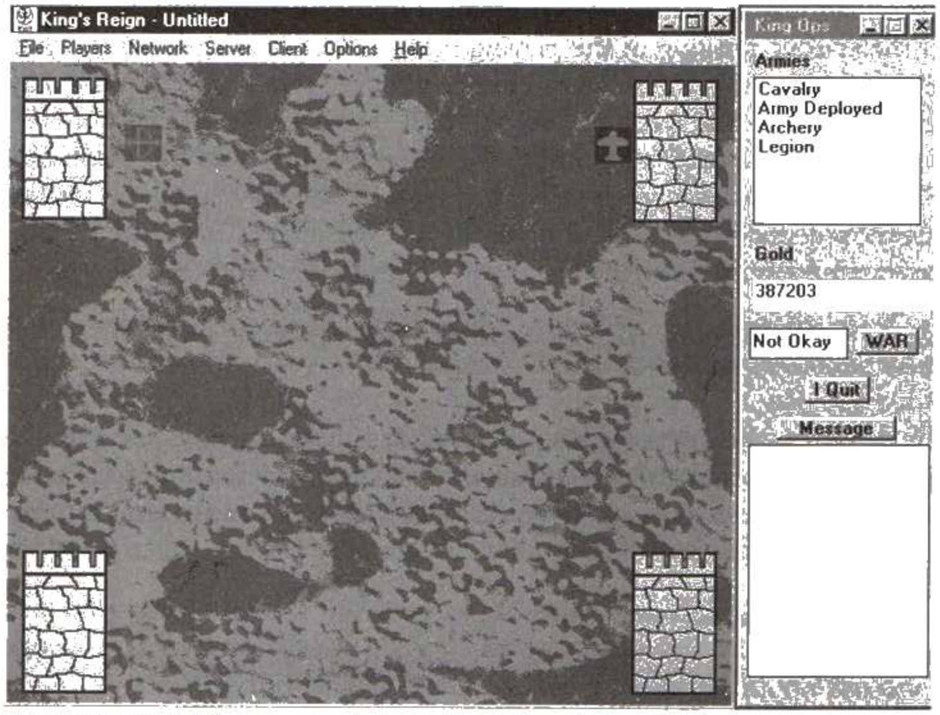


Figure 9.10 *Adding a new playing piece to the playing field.*

able to move a playing piece not only on the player's screen, but on all of the opponent's screens as well.

There are two different ways that we can move the playing piece on the opponent's screens. The first is transmitting each and every move the mouse makes. There are two problems with this approach. The first problem is that most of the moves with the mouse only occur once, because the movement speed of the playing pieces is so small. You cannot move a playing piece from one side of the playing field to the other in a single mouse click. The second problem is that the number of packets being transmitted can be overwhelming for the server. The flyer playing piece will transmit a number of packets during a single move. When four players are moving flyers, things begin to slow down and a loss of sync occurs for a moment during the game. Thus, one player will see the playing piece move but the other players will have a delay in viewing the movement.

The second way to transmit the movement of the playing pieces is to show just the starting position of the playing piece and the ending location when the user releases the mouse button during a move. This means that the only communication that we need to perform is when the OnLButtonUp() method. We will send a packet to the other players with the command SPRITE_STOP, our *who_am_i* value, the value of the playing piece that has been moved, and the X, Y location of the new position of the playing piece. The statement we will use is

```
SendInfo("sprite_stop", playing_pieces[who_am_i][dragging]->mX,
playing_pieces[who_am_i][dragging]->mY, dragging, "");
```

This statement should be placed in the OnLButtonUp() method just after the ReleaseCapture command. Of course we will need some code to handle the command once a player receives it. Add the following code to the OnCommand list of Ifs.

```
else if (strcmp(command, "sprite_stop") == 0)
{
    playing_pieces[player1][sprite2]->MoveTo(GetDC(), player2, sprite1);
}
```

The code takes the information that was sent by the packet and moves the specific playing piece to its new location. Once all of the players receive this packet, all of the playing pieces will be in the same location.

War and the Token

After moving playing pieces around the field, players will eventually want to commit to war on another playing piece. We have somewhat prepared for handling war by creating highlighted playing pieces and creating a war engine. Now we will get into the details of war. Once we are finished with this section, the game is finished in a rough stage. It will be fully playable by up to four players. There are three aspects to a war: token assignment, playing piece designation, and war outcome.

Token Assignment

The structure of the game dictates that allowing more than one person to start a war at the same time causes more problems than it solves. For this reason, we are going to use a war token to allow each player a turn in starting a war. The token will be passed from player to player. The player who has the token will be allowed to select two playing pieces to be used in a war. The first playing piece selected must be one of the player's own pieces.

In this section, we are only going to deal with the mechanism that sends the token from player to player. The idea is to have a token that indicates that the current player is able to begin selecting playing pieces for a war. After some amount of time, the token is packaged up and sent to another player. This player is able to select playing pieces for war before the token is given to another player. In order to start things out correctly, only one of the players in the game is able to start the token. We will use the server player for this task.

The only problem with this type of setup is determining when the token should be move to another player. Thanks to routines available to a Windows programmer, this isn't as hard a task as it might seem. We can use an internal timer provided by the Windows system to send our game a message when a timer expires. We can set the timer to any value that we wish. In this game, we will give each player two seconds to select a playing piece for movement. So that the players will know whether or not they have the token, we will use the empty edit control on the operations window. In this window we will put the words: "Okay For" when the player can start a war and "Not Okay" otherwise. We already have methods in the operations class for displaying these messages.

First we will concern ourselves with the server. The most appropriate time to start the war token will be when the server finds that all clients have registered. Look in the OnMsgAccept() method in the netwavw.cpp file and you will find the code

```
if (the_server->total_connected == total_players-1)
   {
    WSAAsyncSelect(the_server->socket, m_hWnd, 0, 0);
    SendInfo("game_ready", 0, 0, 0, "");

    players[0]->ready_for_game = TRUE;

    MessageBox("All Connected. Begin Game", "Go!", MB_OK);
}
```

At the end of this code, add the following code:

```
war_button = 0;
    war_timer = SetTimer(2, 2000, NULL);

    ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
```

This is the code that will start the war token for the server player. The first line of code sets a variable called *war_button* to the value of the current holder of the war token. We will use this variable when the user tries to select a playing piece for war. Only if *war_button* is equal to our *who_am_i* value will the code allow a playing piece to be highlighted. Before we go any further, we need to declare this variable in the header file of our view class. The code is

```
int war_button;
```

The second line of code starts one of the Windows timers. The first parameter is the timer number to use. We are using another timer, 1, for something else that you will see in Chapter 10. The second parameter is the number of milliseconds to start the timer out at. A value of 2000 represents two seconds. The function SetTimer() returns an identifier for this timer. The value is stored in the variable *war_timer*. This variable also needs to be declared in the view object class as

```
int war_timer;
```

The third line of code sends a message to the operations window telling the window to display the "Okay For" message.

After the code has executed, a timer is running. The timer will run until it expires. At its expiration, the message WM_TIMER will be sent to our application. This is one message that we don't have to add to the system manually. To add the message, go into ClassWizard and Select the class CNetwar2View. Now click on the CNetwar2View object ID. In the right list box, you will find a message called WM_TIMER. Click on this message and click the button Add Function. This will add a message handler for the timers that we use in the game.

OnTimer() Method

The message handler is called OnTimer(UINT nIDEvent). The parameter that is sent to the function is particularly important. The value in the variable *nIDEvent* will be equal to one of the timers that have been set in the game. In other words, you might set three timers running during the course of the game. All of these timers will send the WM_TIMER message when they expire. It is your responsibility to compare the variable *nIDEvent* with the values that the timers returned when started with the SetTimer() function.

In the code so far, the war token timer is the only timer started. We put the value of the timer in the variable *war_timer*. Add the following code to the OnTimer() method:

```
if ((nIDEvent == (UINT)war_timer) && (current_select == 0))
 {
   KillTimer(war_timer);

   if (who_am_i == (total_players-1))
    war_button = 0;
   else
    war_button = who_am_i + 1;

   SendInfo("war_button", war_button, 0, 0, "");
 ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);
 }
```

As was just mentioned, the first thing we have to do is check to make sure that the timer that sent the WM_TIMER message was the war timer. In the same IF statement, we check to see if the player has selected any playing pieces for a war. Recall that the variable *current_select* is used for this purpose. If *current_select* is anything other than 0, then we know that the player has selected a piece.

One thing you should know about the timers. If you don't manually stop the timer, it will continue to send the WM_TIMER after time expires again. The timers automatically reload themselves after sending a WM_TIMER message.

Back to the code. If the IF statement is TRUE, then we enter the block of code. The first thing the code does is call KillTimer(). This function will stop the timer and remove it permanently. The next few lines determine which of the players the token should go to next. It checks to see if we are the last player. If this is the case, the token must return to player 0. Otherwise, the code just sends the token to the next player.

The code sends the token to the next player. Well, kind of. The SendInfo command sends a packet to *all* players in the game. The code has put the player the token should go to in the packet. Our game has no way of sending a packet to just one player. The command in the packet with the token is called war_button. We will have to solve this little problem in the OnCommand() method.

You will notice one thing about this code. If the war timer has expired and the player is currently in the middle of a war, the *current_select* variable will not be 0, so nothing is done with the timer. This means the timer will expire again. It might expire a couple of times before the war sequence has done its job. During all of these expiration periods, our IF statement will evaluate to FALSE. Only when the war is over will the token be sent to another player. We don't have to do anything once the war is over.

For the code to compile, you will need to add the declaration

```
int war_button;
```

to the view class.

OnCommand() Method

When all of the players receive the packet with the war_button command in it, we want only the player who is supposed to receive the token to actually do anything with the packet. This is easy to do, since the player who is to receive the token has its *who_am_i* value in the packet information. Add the following code to the OnCommand() function.

```
else if (strcmp(command, "war_button") == 0)
   {
    war_button = player2;
    if (war_button == who_am_i)
    {
        ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
        war_timer = SetTimer(2, 2000, NULL);
    }
  else
     ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);
  }
```

The first things that the code does is set the variable *war_button* to the field *player2*. This is to guarantee that each of the players knows exactly who has the token. Next, the code checks to see if the value in *war_button* is equal to the machine's *who_am_i* value. If this is true, which it can only be for *one* player, the code displays the "Okay For" message for the player with the token and starts the timer. All of the other players will fall through to the *else* statement and display the "Not Okay" message.

That's it. We have a token that starts with the server and stays there for two seconds. After two seconds and no activity from the player, the token is passed to the next player.

Playing Piece Designation

All of the players are adding and moving playing pieces around the playing field. A war token is being passed from player to player. Once the token arrives at a player, that player can double-click on any of his or her playing pieces to start a war. The player must then double-click on one of their opponent's playing pieces and click on the WAR button to begin a war.

The only time players can click on their playing pieces is when they have a token. We need to change some of the code in the LButtonDblClk() method. We also need to tell the other players to highlight the playing pieces that we have clicked on.

Replace all of the code in the LButtonDblClk() function with the following code

```
CRect* SpriteRect;
  int i, j;

 if (war_button == who_am_i)
 {
  for(i=0;i<total_players;i++)
  {
      for(j=0;j<total_pieces[i];j++)
      {

       SpriteRect = new CRect(playing_pieces[i][j]->mX, playing_pieces[i][j]->mY,
           playing_pieces[i][j]->mX+playing_pieces[i][j]->mWidth,
playing_pieces[i][j]->mY+playing_pieces[i][j]->mHeight);

       if (SpriteRect->PtInRect(point))
       {

        if ((current_select == 1) && (i == who_am_i) && (first_clicked == j))
        {
         current_select = 0;
         SendInfo("sprite_unhi", i, j, 0, "");

         playing_pieces[i][j]->ReplaceBitmap();
         playing_pieces[i][j]->war = FALSE;

         Invalidate(TRUE);
         return;
        }


        if ((current_select == 0) && (i == who_am_i) && (playing_pieces[i][j]->war ==
FALSE))
        {
         first_clicked = j;
         current_select = 1;

         SendInfo("sprite_high", i, j, 0, "");

         playing_pieces[i][j]->ReplaceBitmap();
         playing_pieces[i][j]->war = TRUE;
         Invalidate(TRUE);
         return;
        }


        if ((current_select == 1) && (i != who_am_i) && (playing_pieces[i][j]->war ==
FALSE))
        {
         SendInfo("sprite_high", i, j, 0, "");

         second_clicked_player = i;
         second_clicked_icon = j;

         playing_pieces[i][j]->war = TRUE;

         current_select = 2;
```

```
          playing_pieces[i][j]->ReplaceBitmap();
          Invalidate(TRUE);
          return;
        }
      }
    }
  }
}
```

Before we discuss the code, add the following declarations to our view class.

```
int     first_clicked;
int     second_clicked_player,
          second_clicked_icon;
```

The code is very similar to the code we had before we implemented the war token. The very start of the code has a test that makes sure we are the owner of the war token. This is the case when the variable *war_button* is equal to our *who_am_i* variable. If we have the token, we are allowed to highlight a playing piece.

Since we are in the LButtonDblClk() method, we know the player has double-clicked somewhere on the playing field. The first job of the code is to find out if the double-click was performed on one of the playing pieces. Two loops are used. The outer loop runs through all of the players in the game, and the inner loop runs through all of the playing pieces of the current player. The area of each playing piece is constructed and a test is made using the method PtInRect() to find out if the cursor is on a playing piece.

When it is detected that the cursor is on a playing piece, the code makes three tests. The first test determines if the user is double-clicking on one of his or her own playing pieces to remove a highlight. In other words, the player double-clicked on one of his or her playing piece previous to this double-click. The first double-click highlighted the playing piece for war. By double-clicking on the same playing piece a second time, the highlight is removed and the playing piece is removed from a possible war. Players can only do this if they selected the playing piece first. If another player highlighted the playing piece, the owner of the playing piece cannot remove the highlight.

We know that the playing piece had been previously highlighted in two ways. The first is that the variable *current_select* is set to 1. This indicates the player has clicked on one of his or her playing pieces, but not necessarily this playing piece. The second way be know this is the highlighted playing piece is through the variable *first_selected*. This variable is set to the playing piece number when the piece is first highlighted. If both of these evaluations are true and the playing piece is currently one of ours, we enter the block of code.

The first thing that happens in the code is the variable *current_select* is set back to 0. This means that if the player does not click on another playing piece quickly, the war token will be passed to another player. The second line of code sends a packet to all of the other players with the command SPRITE_UNHI and the sprite number and owner of this playing piece. This command causes the playing piece to be unhighlighted on all of other players' machines. We will see this code in the next section. The next line of code replaces the bitmap with the normal bitmap on our own machine. The fourth line of code sets the playing piece's *war* member variable to FALSE. Finally, the screen is redrawn and control returns from this method.

If any of the tests in the first IF statement fails, control moves to the second IF test. This test is used to highlight a player's playing piece for the first time. In order to highlight the playing piece for war, several conditions have to be met. The *current_select* variable has be set to 0, indicating that no other playing pieces are highlighted, the playing piece has to be owned by us, and the playing piece cannot have its *war* member variable set to TRUE. If all of these conditions are met, the playing piece will be highlighted as our playing piece in a potential war.

The code starts with the variable *first_selected* being set to the index value of this playing piece. We do this so we can remember which one of our playing pieces is being used in a war. Next, the variable *current_select* is set to 1, indicating that one of the two playing pieces for a war has been selected. The code sends a packet to all of the other players with the command SPRITE_HIGH and the owner/index of the playing piece to highlight. The playing piece is highlighted on our machine by replacing its normal bitmap with a highlighted bitmap. The *war* member variable is set to TRUE, the screen is redrawn, and control is returned from this function. We now have our playing piece set for the war.

The third test is executed when the first two fail. This test is used to select an opponent's playing piece for war. The variable *current_select* has to be set to 1, the playing piece cannot be one of ours, and it cannot be in another war. If these conditions are met, the playing piece is highlighted on all of the other machines. Next, the owner and index of the playing piece are saved for use by the war engine. The *war* member variable is set to TRUE, the bitmaps switched, screen redraw, and the variable *current_select* is set to 2. This value means two playing pieces are selected for a war and a war can be started at any time. A war is started when the user clicks on the WAR button on the operations window.

Highlight Commands

There are two additional commands that we have to add to the OnCommand() function. These commands are SPRITE_HIGH and SPRITE_UNHI. The code to add is:

```
else if (strcmp(command, "sprite_high") == 0)
  {
   playing_pieces[player2][sprite1]->war = TRUE;
   playing_pieces[player2][sprite1]->ReplaceBitmap();
   Invalidate(TRUE);
  }
  else if (strcmp(command, "sprite_unhi") == 0)
  {
   playing_pieces[player2][sprite1]->war = FALSE;
   playing_pieces[player2][sprite1]->ReplaceBitmap();
   Invalidate(TRUE);
  }
```

The code simply sets the *war* member variable of the appropriate playing piece to TRUE or FALSE depending on the command and replaces the bitmap. Since the ReplaceBitmap() function is a toggle, both of the commands use the same statement.

New Code

In order to see the highlighting in action, the code up to this point can be found in the /netwar2/chapt9/e directory. Execute the code and start double-clicking on playing pieces. Once you double-click on one of your pieces and one of an opponent's pieces, they will remain highlighted. We need the WAR button operational to turn the highlights off.

War Outcome

It's time to put it all together and allow a war to be played out. Once a player has the war token, he or she can double-click on one of his or her playing pieces and then find another opponent's playing piece for a war. After both playing pieces are clicked on, the player clicks the WAR button on the operations window. The war engine is put into use, and the outcome of a war is determined. Based on the results returned from the war engine, things may have to occur to each of the playing pieces.

Right at the beginning, things should be set up so that a playing piece on one side of the playing field cannot attack a playing piece on the other side. We will use a distance calculation to determine the distance between the two highlighted pieces and see if the war can take place.

The operations window is already set up to send the view object the message WM_WAR_BUTTON when the WAR button is clicked. We need to add a message macro to the map in the netwavw.cpp file. The line is

```
ON_MESSAGE(WM_WAR_BUTTON, OnWar)
```

The function that will handle the message is called OnWar(). The code is

```
long CNetwar2View::OnWar(UINT wParam, LONG lParam)
{
 long length;
 int i, defense, offense;
 CRect* rect;
 CPoint pnt;

 if (war_button == who_am_i)
 {

 if (current_select == 2)
 {

  current_select = 0;
  playing_pieces[second_clicked_player][second_clicked_icon]->war = FALSE;
  playing_pieces[who_am_i][first_clicked]->war = FALSE;
```

```
    // remove highlight on remote machines
    SendInfo("sprite_unhi", who_am_i, first_clicked, 0, "");
    SendInfo("sprite_unhi", second_clicked_player, second_clicked_icon, 0, "");

    // remove highlight on our machine
    playing_pieces[who_am_i][first_clicked]->ReplaceBitmap();
    playing_pieces[second_clicked_player][second_clicked_icon]->ReplaceBitmap();

    length =
(int)sqrt(pow(playing_pieces[second_clicked_player][second_clicked_icon]->mX-
playing_pieces[who_am_i][first_clicked]->mX, 2)
                        +
pow(playing_pieces[second_clicked_player][second_clicked_icon]->mY-
playing_pieces[who_am_i][first_clicked]->mY, 2));

    if (length < 100)
    {
      defense = playing_pieces[second_clicked_player][second_clicked_icon]->defense;
      offense = playing_pieces[who_am_i][first_clicked]->offense;

      switch(war_matrix->get_battle_result(offense, defense))
      {
        case NO_DAMAGE:
                        break;

        case HALF_DAMAGE:
                    if (playing_pieces[second_clicked_player][second_clicked_icon]-
>offense > 0)
                    {
playing_pieces[second_clicked_player][second_clicked_icon]->offense =
playing_pieces[second_clicked_player][second_clicked_icon]->offense / 2;
                    }
                    if
(playing_pieces[second_clicked_player][second_clicked_icon]->defense > 0)
                    {

playing_pieces[second_clicked_player][second_clicked_icon]->defense =

playing_pieces[second_clicked_player][second_clicked_icon]->defense / 2;
                    }

                        SendInfo("sprite_offe", second_clicked_player,
second_clicked_icon,
        playing_pieces[second_clicked_player][second_clicked_icon]->offense, "");
                        SendInfo("sprite_defe", second_clicked_player,
second_clicked_icon,
        playing_pieces[second_clicked_player][second_clicked_icon]->defense, "");

                    if
((playing_pieces[second_clicked_player][second_clicked_icon]->defense == 0) &&
(playing_pieces[second_clicked_player][second_clicked_icon]->offense == 0))

                {
playing_pieces[second_clicked_player][second_clicked_icon]->active = FALSE;
                  Invalidate(TRUE);
                  players[0]->kills++;
                }
                break;

        case ATTACKER_DAMAGE:
                  if (playing_pieces[who_am_i][first_clicked]->offense > 1)
                  {
                    playing_pieces[who_am_i][first_clicked]->offense =
                                playing_pieces[who_am_i][first_clicked]-
>offense / 2;
                }

                      if (playing_pieces[who_am_i][first_clicked]->defense > 1)
                        {
                          playing_pieces[who_am_i][first_clicked]->defense =
                                playing_pieces[who_am_i][first_clicked]-
>defense / 2;
                }



                          SendInfo("sprite_offe", who_am_i, first_clicked,
          playing_pieces[who_am_i][first_clicked]->offense, "");
                      SendInfo("sprite_defe", who_am_i, first_clicked,
                                playing_pieces[who_am_i][first_clicked]-
>defense, "");

                            if ((playing_pieces[who_am_i][first_clicked]-
>offense == 0) &&
                                (playing_pieces[who_am_i][first_clicked]-
>defense == 0))
                            {
                              playing_pieces[who_am_i][first_clicked]-
>active = FALSE;
                              Invalidate(TRUE);
                              SendInfo("sprite_kill", 0, 0, 0, "");
                            }
                          break;

      case BOTH_DAMAGE:
     if (playing_pieces[second_clicked_player][second_clicked_icon]->offense > 0)
     {
      playing_pieces[second_clicked_player][second_clicked_icon]->offense =
            playing_pieces[second_clicked_player][second_clicked_icon]->offense / 2;
     }
```

```
(playing_pieces[second_clicked_player][second_clicked_icon]->offense == 0))

                {
playing_pieces[second_clicked_player][second_clicked_icon]->active = FALSE;
                   Invalidate(TRUE);
                   players[0]->kills++;
                }
                break;

        case ATTACKER_DAMAGE:
                   if (playing_pieces[who_am_i][first_clicked]->offense > 1)
                   {
                      playing_pieces[who_am_i][first_clicked]->offense =
                                 playing_pieces[who_am_i][first_clicked]-
>offense / 2;
                }

                         if (playing_pieces[who_am_i][first_clicked]->defense > 1)
                            {
                              playing_pieces[who_am_i][first_clicked]->defense =
                                 playing_pieces[who_am_i][first_clicked]-
>defense / 2;
                }



                              SendInfo("sprite_offe", who_am_i, first_clicked,
           playing_pieces[who_am_i][first_clicked]->offense, "");
                         SendInfo("sprite_defe", who_am_i, first_clicked,
                                 playing_pieces[who_am_i][first_clicked]-
>defense, "");

                                if ((playing_pieces[who_am_i][first_clicked]-
>offense == 0) &&

                                    (playing_pieces[who_am_i][first_clicked]-
>defense == 0))

                                {
                                   playing_pieces[who_am_i][first_clicked]-
>active = FALSE;

                                   Invalidate(TRUE);
                                   SendInfo("sprite_kill", 0, 0, 0, "");
                                }
                              break;

      case BOTH_DAMAGE:
     if (playing_pieces[second_clicked_player][second_clicked_icon]->offense > 0)
     {
      playing_pieces[second_clicked_player][second_clicked_icon]->offense =
            playing_pieces[second_clicked_player][second_clicked_icon]->offense / 2;
     }
```

```
   if (playing_pieces[second_clicked_player][second_clicked_icon]->defense > 0)
    {
     playing_pieces[second_clicked_player][second_clicked_icon]->defense =
          playing_pieces[second_clicked_player][second_clicked_icon]->defense / 2;
    }

   SendInfo("sprite_offe", second_clicked_player, second_clicked_icon,
    playing_pieces[second_clicked_player][second_clicked_icon]->offense, "");
   SendInfo("sprite_defe", second_clicked_player, second_clicked_icon,
    playing_pieces[second_clicked_player][second_clicked_icon]->defense, "");

    if ((playing_pieces[second_clicked_player][second_clicked_icon]->defense == 0)
&&

 (playing_pieces[second_clicked_player][second_clicked_icon]->offense == 0))
                {

 playing_pieces[second_clicked_player][second_clicked_icon]->active = FALSE;
                 Invalidate(TRUE);
                 players[0]->kills++;
                }


                if (playing_pieces[who_am_i][first_clicked]->offense > 1)
                {
                          playing_pieces[who_am_i][first_clicked]->offense =
                          playing_pieces[who_am_i][first_clicked]->offense / 2;
                            }

                       if (playing_pieces[who_am_i][first_clicked]->defense > 1)
                       {
                            playing_pieces[who_am_i][first_clicked]->defense =
                          playing_pieces[who_am_i][first_clicked]->defense / 2;
                            }


                SendInfo("sprite_offe", who_am_i, first_clicked,
                     playing_pieces[who_am_i][first_clicked]->offense, "");
                     SendInfo("sprite_defe", who_am_i, first_clicked,
                          playing_pieces[who_am_i][first_clicked]->defense, "");

                          if ((playing_pieces[who_am_i][first_clicked]->offense
== 0) &&

                            (playing_pieces[who_am_i][first_clicked]->defense ==
0))

                          {
                           playing_pieces[who_am_i][first_clicked]->active =
FALSE;

                           Invalidate(TRUE);
                           SendInfo("sprite_kill", 0, 0, 0, "");
```

```
                                      }
                    break;

            case DESTROYED:
                    playing_pieces[second_clicked_player][second_clicked_icon]->active
= FALSE;
                    SendInfo("sprite_offe", second_clicked_player,
second_clicked_icon, 0, "");
                    SendInfo("sprite_defe", second_clicked_player,
second_clicked_icon, 0, "");
                    Invalidate(TRUE);

                    players[0]->kills++;
                    break;
      }
    }
    else
    {
      MessageBox("Too far away to launch an attack", "Sorry...", MB_OK);

      Invalidate(TRUE);
      }
    }
  }
  return 1;
}
```

Now, that's a bit of code. The function starts out by making sure that we have the war token. If we don't have the war token, then we shouldn't be clicking on the WAR button. If we do click on the WAR button, the code better not do anything. Next, a test is made of the *current_select* variable. The value needs to be 2 so that the code knows we have selected two playing pieces for the war.

The *war* member variables of the playing pieces are set to FALSE on our machine. A packet is sent to all other players telling them to unhighlight the two playing pieces currently highlighted. If you remember from above, the code for the unhighlight also set the *war* variables to FALSE. Next, we unhighlight the playing pieces on our machine. Now all machines have unhighlighted the playing pieces, and the *war* variables are set to FALSE.

Next, the distance between both of the warring playing pieces is calculated. A test is made of the resulting value. If the value is greater than 100 pixels, a war cannot take place. In this case, a message is displayed to the users letting them know that a war is not going to take place because the playing pieces are too far apart. The screen is redrawn and the function is exited.

If the distance between the playing pieces is less than 100, a war will take place. The code sets two local variables, *offense* and *defense,* to the respective values of the playing pieces involved in the war. The outcome of the war is received with the statement to the war engine

```
war_matrix->get_battle_result(offense, defense)
```

The value returned from the get_battle_result() command is used in a *switch* statement. There are only five possible outcomes.

No Damage

In the case of no damage, there is nothing to do but return from the function.

Half Damage

When the half damage result is given, the attacked piece must be penalized half of its offense and defense points. We have to make sure that not only are the points deducted from the attacked playing piece on our machine, but all of the opponents must be made aware of the new points as well. It should be noted that a player piece can have a offense or defense value of 0.

The code in this case checks to make sure that first the offense points of the attacked are greater than 0. If the offense points are greater than 0, then they are divided by 2. The system will automatically round the resulting value.

The same thing is done for the defense points. Now the change in offense and defense points has been made on our machine only. Because of this, two packets are sent to all opponents. The first packet has the command SPRITE_OFFE and includes the new offense points and the owner/index of the playing piece the offense points should be applied to. The second packet has the command SPRITE_DEFE and includes the same information, except it includes the new defense points.

After the packets are sent, the code checks to see if both the offense and defense points are zero. If this is the case, the playing piece has been destroyed and needs to be removed from the playing field. This is performed by setting the playing piece's *active* member variable to FALSE and executing the Invalidate(TRUE) function call. The last line of code in this block is an incrementing of the *kills* member variable of the current player.

You might be asking yourself about that destroyed playing piece and the other opponents. We don't send anything to them letting them know that the playing piece is no longer valid. All of this is handled in the SPRITE_OFFE and SPRITE_DEFE commands. The code that handles these commands checks and will destroy the playing piece if both variables are zero.

Attacker Damage

The attacker damage case is the same as the half damage case except the points are taken away from the playing piece that started the war.

Both Damage

The both damage case is the combination of the half damage and attacker damage cases. Both the attacker and attacked are deducted points.

Destroyed

Only the attacked playing piece can be destroyed in a war. The offense and defense points are set to zero, and the piece is removed from the playing field.

OnCommand()

The war routines used two additional packet commands that aren't in our current OnCommand() function. Add the following code.

```
else if (strcmp(command, "sprite_offe") == 0)
   {
    playing_pieces[player2][sprite1]->offense = sprite2;
    if (playing_pieces[player2][sprite1]->offense == 0 &&
       playing_pieces[player2][sprite1]->defense == 0)
    {
     playing_pieces[player2][sprite1]->active = FALSE;
     Invalidate(TRUE);
    }
  }
  else if (strcmp(command, "sprite_defe") == 0)
  {
   playing_pieces[player2][sprite1]->defense = sprite2;
  if (playing_pieces[player2][sprite1]->offense == 0 &&
      playing pieces[player2][sprite1]->defense == 0)
    {
     playing pieces[player2][sprite1]->active = FALSE;
     Invalidate(TRUE);
    }
  }
```

The new code handles the SPRITE_OFFE and SPRITE_DEFE commands. The appropriate variable is set in both cases, and a check made of both variables. If the variables are zero, the playing piece is removed from the playing field of this player.

The Code

The code for the game to this point can be found in the directory /netwar2/chapt9/f. This is a complete game. You can connect up to four players, purchase equipment, add and move the equipment around the playing field, and perform wars against each other. You will notice one thing, however: You don't really know who is doing war against whom. Even more important, you don't know the outcome of the war. It's time to add some messaging to the game.

Messaging

The purpose of our messaging system is to allow the game system to display messages to the players. The main information displayed will concern a war between players and the outcome. All of the messages will appear in the message list box on the operations window. When we designed the window, we defined the variables and functions necessary for displaying messages.

The sequence that must be used to add a message to the list box is

> copy the message to the member variable *ops_dlg->new_message*,
>
> send the message WM_OPERATIONS_UPDATE_MESSAGE to the operations window.

This sequence of operations will display a message on our operations window. During a war, the system will display who is in the war and its outcome on all operations windows. This sounds like something a packet can handle. We will use the packet command PRINT_STR to display any message on a player's message list box. When this command is sent to another player, a message number is sent as well. The OnCommand() function will take this number and display a particular message in the message list box.

We will start with the war. Add the following code.

```
sprintf(ops_dlg->new_message, "War - Player %d and %d", who_am_i,
second_clicked_player);
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);

SendInfo("print_str", 1, who_am_i, second_clicked_player, "");
```

just after the statements

```
if (length < 100)
{
```

in the OnWar() function.

The message that is being displayed is "War - Player 1 and 2" with the player numbers in the appropriate places. The message is copied to the *new_message* member variable of the operations window. The third line of code sends a packet to the other players with a message number 1 and the player numbers of the two players in the game.

The packet will need to be processed in the OnCommand() function. Add the following code to the function.

```
else if (strcmp(command, "print_str") == 0)
   {
      MessageBeep(-1);
      switch(player2)
      {
         case 1: sprintf(ops_dlg->new_message, "War: %d and %d", sprite1, sprite2);
                 ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
                 break;
       }
      }
```

When the PRINT_STR command is received by the other players in the game, a beep will occur to let them know that a message has been added to the message list box. A *switch* statement is used to print the appropriate message, based on the message number sent with the packet. The message is displayed in the player's message list box just as it is in the message list box of the player who sent the message.

War Outcome Messages

The message above was used to display a message about the two players currently in a war. We also want to display the outcome of the war to all of the players. In the OnWar() function, you need to add the following code to the appropriate *case* statements:

In the *case* statement **NO_DAMAGE**

```
sprintf(ops_dlg->new_message, “No Damage”);
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
SendInfo(“print_str”, 2, 0, 0, “”);
```

In the *case* statement **HALF_DAMAGE**

```
sprintf(ops_dlg->new_message, “Half Damage”);
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
SendInfo(“print_str”, 3, 0, 0, “”);
```

In the *case* statement **ATTACKER_DAMAGE**

```
sprintf(ops_dlg->new_message, “Attacker Damaged”);
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
SendInfo(“print_str”, 5, 0, 0, “”);
```

In the *case* statement **BOTH_DAMAGE**

```
sprintf(ops_dlg->new_message, “Both Damaged”);
ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
SendInfo(“print_str”, 4, 0, 0, “”);
```

In the *case* statement **DESTROYED**

```
sprintf(ops_dlg->new_message, “Destroyed”);
ops_lg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
SendInfo(“print_str”, 6, 0, 0, “”);
```

Each of the pieces of code displays the outcome of the war on the player's operation window first and then sends a message to each of the other players. We need to add five additional message numbers to the *switch* statement in the OnCommand() function and the PRINT_STR command. The code to add is:

```
case 2: sprintf(ops_dlg->new_message, "No Damage");
          ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
          break;

case 3: sprintf(ops_dlg->new_message, "Half Damage");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
          break;

case 4: sprintf(ops_dlg->new_message, "Both Damaged");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
          break;

case 5: sprintf(ops_dlg->new_message, "Attacker Damaged");
          ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
          break;

case 6: sprintf(ops_dlg->new_message, "Destroyed");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
          break;
```

Anywhere that you would like to add messages to the system, you can do it using the above statements.

Summary

The game is complete. All of the fundamental operations are in place. All that is left to add is the small things that make a game more professional looking. In addition, we need to add support for modem operations.

Chapter 10

Finishing the Game

Adding socket support to our game gives us the ability to play with other opponents throughout the world. As you play the game yourself, you might find several places where additional instruction or features need to be added. This chapter will deal with those additions:

- Messaging between players
- Adding a QUIT button
- Extra screen refreshing
- New Game
- Game ready message
- Server waiting message
- Lost war token recovery
- Gold Mine

Messaging Between Players

When we designed the operations window, we put in the components necessary for a messaging system between the players. A player can send a message by clicking on the Message button on the operations window. When this occurs, a message dialog box appears as in Figure 10.1.

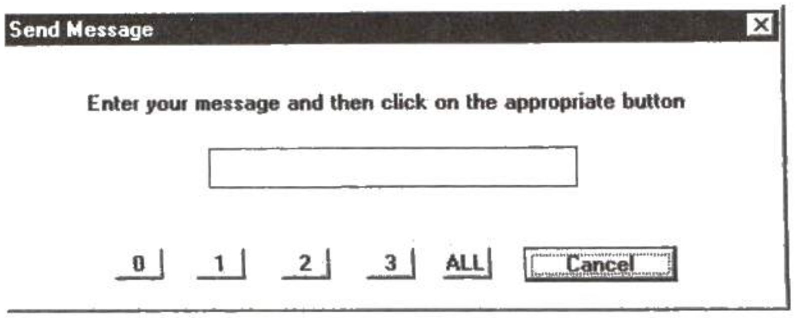


Figure 10.1 *Sending a message.*

As you can see, the message dialog box allows a message to be entered and sent to either one player or all players. Remember that the packet that we are using to send information between players has a final field that allows a 32-character string. This is what we will use to send the message to the other player(s).

The first step is to create the message dialog box. The box should be created something like that shown above. The class name for the box is CMessaging. You will need to add the following controls:

Dialog Box ID:	IDD_MESSAGE_INPUT		
Edit Control:	IDC_MESSAGE_INPUT		
Button:	IDC_SEND_MESSAGE_1	labeled:	0
Button:	IDC_SEND_MESSAGE_2	labeled:	1
Button:	IDC_SEND_MESSAGE_3	labeled:	2
Button:	IDC_SEND_MESSAGE_4	labeled:	3
Button:	IDC_SEND_MESSAGE_ALL	labeled:	ALL
Button:	IDCANCEL	labeled:	Cancel

In ClassWizard, you will need to create message maps for all of the buttons. You will also need a member variable for the edit control: IDC_MESSAGE_INPUT. The variables should be named *m_message_input*.

This does everything we need for the dialog box itself. Now we need to add code for the remainder of the messaging system. Open the file messagin.cpp and add the following code to the specific functions.

```
void CMessaging::OnSendMessage1()
{
   sendto = 1;
   CDialog::OnOK();
}
void CMessaging::OnSendMessage2()
{
   sendto = 2;
   CDialog::OnOK();
}
void CMessaging::OnSendMessage3()
{
   sendto = 3;
   CDialog::OnOK();
}
void CMessaging::OnSendMessage4()
{
   sendto = 4;
   CDialog::OnOK();
}
void CMessaging::OnSendMessageAll()
{
   sendto = 5;
   CDialog::OnOK();
}
void CMessaging::OnCancel()
{
   sendto = 0;
   CDialog::OnCancel();
}
```

When the Message dialog box is activated, players will enter their messages in the edit control. Once the message is ready to be sent, the player clicks on one of the available buttons. In all cases, the button message handlers set a member variable *sendto* equal to a specific number. When the player cancels the message, the value of 0 is placed in this variable. When all of the players are to receive the message, the value of 5 is placed in the variable. A message sent to a specific player has the player's number placed in the variable.

Clicking on any of the buttons also causes the dialog box to be removed from the screen. The function that displayed the dialog box can look at the *sendto* variable to determine who to send the message to and the variable *m_message_input* for the message itself. The variable *sendto* will need to be declared in the messagein.h file as an integer.

When the user clicks on the Message button, the view will need to know about it. Open the operatio.cpp file and update the following code:

```
void operations::OnOperationsMessageButton()
{
   if (m_pView != NULL)
    m_pView->SendMessage(WM_MESSAGE_BUTTON, IDOK);
}
```

Now open the netwar2.h file and add the code:

```
#define WM_MESSAGE_BUTTON WM_USER + 106
```

This code will send the message WM_MESSAGE_BUTTON to the view object when the message button is clicked. We will need a function and message macro to handle the message. In the netwavw.h file, add the function prototype:

```
long OnMessageButton(UINT wParam, LONG lParam);
```

Open netwavw.cpp, add the message macro

```
ON_MESSAGE(WM_MESSAGE_BUTTON, OnMessageButton)
```

and the function itself:

```
long CNetwar2View::OnMessageButton(UINT wParam, LONG lParam)
{
   CMessaging pDlg;
   char buffer[45];

   pDlg.DoModal();

   sprintf(buffer, "%s", pDlg.m_message_input);
   if (pDlg.sendto > 0)
   SendInfo("msg", pDlg.sendto-1, who_am_i, 0, buffer);

  return 1;
}
```

The function is simple. An object is created for the Cmessaging class. The dialog box is displayed. Once the player clicks on one of the buttons on the dialog box, control is returned to this function. If any button other than Cancel is clicked, the message is packaged and sent to the other players. Notice that all of the players will receive the packet. However, only the player with the *who_am_i* variable equal to the first parameter will display the message.

We need the code for the OnCommand() function. Add the following code to the function:

```
else if (strcmp(command,"msg") == 0)
   {
    if ((player2 == who_am_i) || (player2 == 4))
    {
     sprintf(ops_dlg->new_message, "%d: %s", sprite1, msg);
     ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
    }

    MessageBeep(-1);
   }
```

The command used for the messaging system is MSG. When received, the code will check to see if all of the players are to receive the message or if this player is the only one. The message is copied to the appropriate member variable of the operations window, and a message is sent to the operations class to display the message. Finally, a beep is made to signal the player that a message has arrived.

Miscellaneous

The following sections outline several small things that need to be added to the game. These additions make the game more playable.

I Quit

If a player is getting stomped, he or she might want to quit the game entirely, so we'll put a Quit button on the operations window. When users click on the Quit button, their gold should be set to zero and all of their playing pieces removed from the playing field.

Update the following code in the netwavw.cpp file.

```
long CNetwar2View::OnQuitButton(UINT wParam, LONG lParam)
{
 int i;

 for(i=0;i<total_pieces[who_am_i]; i++)
  playing_pieces[who_am_i][i]->active = FALSE;

 total_pieces[who_am_i] = 0;

 sprintf(ops_dlg->new_message, "Player %d has surrendered", who_am_i,
second_clicked_player);
 ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);

 ops_dlg->SendMessage(WM_OPERATIONS_PLAYER_QUITS);

 SendInfo("sprite_quit", who_am_i, 0, 0, "");
 SendInfo("print_str", 7, who_am_i, who_am_i, "");

 Invalidate(TRUE);

 return 1;
}
```

The code starts by setting all of the player's playing pieces' *active* variable to FALSE. The total number of pieces for the player is set to zero. A message is displayed on all of the players' message list boxes letting them know that a player that surrendered. The code uses two packets with commands SPRITE_QUIT and PRINT_STR to finish up the quitting of a player.

We need to add the following code to the OnCommand() function

```
else if (strcmp(command, "sprite_quit") == 0)
 {
   for(i=0;i<total_pieces[player2];i++)
    playing_pieces[player2][i]->active = FALSE;

   total_pieces[player2] = 0;

   Invalidate(TRUE);
 }
```

and add this *case* statement to the PRINT_STR handler in the OnCommand function:

```
case 7: sprintf(ops_dlg->new_message, "%d surrendered", sprite1);
          ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
             break;
```

Since the player's gold pieces have been set to 0, the player cannot purchase more equipment and all current equipment has been removed from the game.

One of the messages that is sent to our game is called WM_OPERATIONS_PLAYER_QUITS. This message tells the operations window to remove all of the armies from the Armies list box and zero out the gold coins message area.

You will need to add the following code to the operatio.h file

```
#define WM_OPERATIONS_PLAYER_QUITS  WM_USER + 107
```

and add a prototype for the message handler

```
long OnPlayerQuits(UINT wParam, LONG lParam);
```

Now open the operatio.cpp file and add a message macro

```
ON_MESSAGE(WM_OPERATIONS_PLAYER_QUITS, OnPlayerQuits)
```

and the following code.

```
long operations::OnPlayerQuits(UINT wParam, LONG lParam)
{
 CListBox* box;
 CEdit* box2;

 box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
 box->ResetContent();

 box2 = (CEdit*)GetDlgItem(IDC_OPERATIONS_AVAILABLE_GOLD);
 box2->SetSel(0, -1, FALSE);
 box2->ReplaceSel("");

 total_armies = 0;
 return 1;
}
```

Extra Invalidate

Occasionally, the main screen does not refresh itself correctly when all of the players are moving their playing pieces at the same time. The screen is refreshed only when a playing piece is put in its final position on the screen. If two players are moving playing pieces in the same basic location, the overlapping playing pieces will not refresh properly. To change this, we can add an additional Invalidate(TRUE) statement each time the player moves a playing piece. Since we don't always need the extra invalidate, we will create a menu item that toggles a Boolean variable. If the variable is TRUE, the extra invalidate is used.

In AppStudio, add a menu entry under the menu item Options called Extra Invalidate. Open ClassWizard and create a function to handle the command the command update messages for the menu item.

Find the new function in the netwavw.cpp file and update the following code

```
void CNetwar2View::OnOptionsExtrainvalidate()
{
     if (DO_EXTRA_INVAL)
       DO EXTRA INVAL = FALSE;
  else
       DO_EXTRA_INVAL = TRUE;
}

void CNetwar2View::OnUpdateOptionsExtrainvalidate(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(DO_EXTRA_INVAL == TRUE);
}
```

We need to declare the *DO_EXTRA_INVAL* variable in the view object file as a Boolean variable. The variable needs to be declared in the constructor as well. Use the statement

```
DO_EXTRA_INVAL = FALSE;
```

The last thing we need to do is put some code in the game to use the extra invalidate. In the function OnMouseMove(), add the following code in the IF statement.

```
if (DO EXTRA_INVAL) Invalidate(TRUE);
```

During game play, you can click on the Extra Invalidate menu entry to activate the code.

New Game

When a player clicks on the menu item for a new game, all of the players need to know. In the function OnFileNew(), add the following statement.

```
SendInfo("game_new", who_am_i, 0, 0, "");
```

This statement will send a packet to the other players letting them know that a new game is being requested. The command is called GAME_NEW, and we need to have a handler in the OnCommand() function. Add the following code to that function:

```
else if (strcmp(command, "game new") == 0)
   {
     ops dlg->SendMessage(WM_OPERATIONS_NEW_GAME, 0, 0L);
     for (i=0;i<total_players; i++)
     {
       for (j=0;j<total_pieces[i];j++)
         delete playing_pieces[i][j];

       total_pieces[i] = 0;
     }
     players[0]->gold_coins = 500000;
     players[0]->kills = 0;
     MessageBox("Game Has Been Reset", "New Game", MB_OK);

     Invalidate(TRUE);
   }
```

The code is much the same as the new game code for the player who clicks on the menu item. The only difference is that all of the players will have a dialog box displayed letting them know about the new game.

Server Wait Message

If you have played the game, you will have already noticed that when you tell the server to wait for connections from clients, there is nothing telling you that this is happening. You just click on the menu item and the server is ready. Let's add a dialog box that will be displayed until all of the clients are connected.

Start AppStudio and create a new dialog box with just a static text control with something in it like:

"Server is waiting for connections from clients."

You don't need to have any buttons in the dialog box, since the code will take care of creating and removing the dialog box. ID the dialog box IDD_SERVER_WAIT. We don't need to create a class for the dialog box because there are no controls in the box.

Open ClassWizard and name the new dialog box CServerWait. Add the declaration

```
CServerWait msg_dlg;
```

to the view object class. Open the netwavw.cpp file and add the following code line to the beginning of the code.

```
#include "serverwa.h"
```

Move down to the function OnWaitingforConnect() and add the code

```
msg_dlg.Create();
```

Now move to the function OnMsgAccept(). In the IF statement that checks for all connections add the code

```
msg_dlg.DestroyWindow();
```

As the server waits for connections from the clients, a dialog box will be displayed. When all of the clients connect, the box will be eliminated.

Game Ready

When all of the clients are attached to the server, a packet is set to all of the players with the command GAME_READY. The purpose of this packet is to let all of the players know that all of the other players are ready. The command code will display a message box with this information.

Add the following code to the OnCommand() function:

```
else if (strcmp(command, "game_ready") == 0)
 {
  war_button = 0;

  ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);

  MessageBox("All Players connected. Begin Game", "Ready to go...", MB_OK);
 }
```

The code also sets up some of the war token information. Notably, the *war_button* variable is set to zero, reflecting the fact that the server will be the first player with the token. In addition, the message "Not Okay" is displayed on the operations window.

War

The war engine works well in the game, assuming that you entered the outcomes fairly in the matrix. There are four additional aspects of war that we must investigate. These are a lost war token, adding a gold mine, giving extra offense points for kills greater than 4, and increasing defense points when users are in their castles.

Lost War Token

During game play, we have a token packet that is being sent to each of the players in the game in succession. Player 0 will start with the token and send it after 2 seconds of no war activity to player 1. This continues until the token returns to player 0 and the sequence starts all over again. Although the TCP/IP mechanism under the Windows socket library is supposed to ensure guaranteed communication between machines, the token sometimes gets lost. When the token is lost, none of the players can start a war. The game is basically dead.

To relieve this situation, we need to come up with a way of restarting the token passing. You will recall that the token is held at each player for 2 seconds. After the 2 seconds have expired, the token is sent to the next player. Now, if there are four players in the game and each player gets the token for 2 seconds, player 0 should receive the token after 8 to 10 seconds. What if we set a timer at the server that expired 10 seconds after the war token is sent to player 1? When the war token is returned to player 0, the timer would be killed. If the timer ever expires, we know the war token has been lost.

This seems like a pretty good plan, but it has a slight problem. When a player receives the war token and clicks on one of his or her pieces, the token is paused until the outcome of the war is known. If the player takes time in selecting an opponent, the timer is sure to expire, indicating that the war token has been lost. In fact, the war token is sitting with the player starting a war.

This actually brings up a good point. Should a player be given a time limit to select a second player and click the war button? Once the player selects a playing piece in the war, a timer could be started for, say, 5 seconds. The player must click on the opponent's playing piece and the WAR button before the 5-second timer expires. Otherwise, the playing pieces are unsolicited and the war token is sent to the next player.

In the original game, this type of restriction is not included. You are welcome to add it. To alleviate the lost token problem, a timer is used, as in the first discussion, with an expiration of 30 seconds. This has shown to be plenty of time and could probably be reduced.

The code for the lost timer is added to the OnTimer() function in the IF statement where the token is passed to player 1. The code in the OnTimer() function now appears as:

```
if ((nIDEvent == (UINT)war_timer) && (current_select == 0))
 {
  KillTimer(war_timer);

  if (who_am_i == (total_players-1))
   war_button = 0;
  else
  {
   war_button = who_am_i + 1;
   if (war_button == 1)
    lost_token_timer = StartTimer(3, 30000, NULL);
  }

  SendInfo("war_button", war_button, 0, 0, "");
  ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);
 }

 if (nIDEvent == (UINT)lost_token_timer)
 {
  KillTimer(lost_token_timer);

  war_button = 0;
  SendInfo("war_button", war_button, 0, 0, "");
  ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
 }
```

You will also need to add the declaration

```
int lost_token_timer;
```

The lost token code was added in two places. First, look at the code at the top, where the *war_button* variable is set equal to the *who_am_i* variable plus 1. The code makes a quick check to see if the new recipient of the war token is supposed to be player 1. If this is the case, we know the server, player 0, is sending the war token to player 1. The code starts a new timer with an expiration time of 30 seconds and sets the variable *lost_token_timer* equal to the returned value.

After this code, a new IF statement has been added. This new IF statement checks to see if the OnTimer() function was called for the lost token timer. If this is true, the lost token timer is killed and the war token is given to the server. Notice that the lost token timer will only be started on the server machine and not on any of the other players' machines.

Gold Mine

To give the game some added detail, a gold mine is going to be placed near the center of the map. The gold mine will dispense ten gold pieces per real-time second. The catch to the gold mine is that it will only dispense the gold when a single playing piece is on it.

Notice that I said ten gold pieces are dispensed every second. Sounds as if we are going to have another timer. The new timer will be set to expire every second. When the timer expires, a check will be made on each machine to see if a playing piece is on the gold mine. A check will also be made for a second playing piece so that the code knows if the gold pieces should be dispensed. Two tasks are before us: starting the timer and responding to it.

First, we will start the gold mine timers. Add the following code to the header file for our view.

```
int gold_mine_timer;
Crect* gold_mine_rect;
```

Now in the OnCommand() function, add the following command code to the GAME_READY command.

```
gold_mine_timer = SetTimer(1, 1000, NULL);
gold mine rect = new CRect(185,185,235,235);
```

This code sets the timer when the GAME_READY command is sent by the server. We also need to start the gold mine timer for the server. Add the same line of code to the OnMsgAccept() function where we have the IF statement that checks for all clients connected to the server.

Now we need to respond to the timer. Add the following code to the OnTimer() function:

```
if (nIDEvent == (UINT)gold_timer)
 {
  one_of_us = FALSE;
  one_of_them = FALSE;

  for(i=0;i<total_players;i++)
  {
   for(j=0;j<total_pieces[i];j++)
   {
    pnt = CPoint(playing_pieces[i][j]->mX, playing_pieces[i][j]->mY);

    if ((gold_mine_rect->PtInRect(pnt)) && (playing_pieces[i][j]->active == TRUE))
    {
     if (i == who_am_i)
      one_of_us = TRUE;
     else
      one_of_them = TRUE;
    }
   }
  }

  if ((!one_of_them) && (one_of_us))
  {
   players[0]->gold_coins += 10;
   ops_dlg->total_gold = players[0]->gold_coins;
   ops_dlg->PostMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);
   return;
  }
 }
```

Also add the following declaration at the beginning of the same function:

```
BOOL one_of_us, one_of_them;
```

As in all of the other timers, the code must check to see if the OnTimer() function was called for the gold_mine_timer. Next the two Boolean variables are set to FALSE. The code enters a double-nested loop. For every playing piece that is on the playing field, a check is made to see if the playing piece is in the gold mine rectangle. If one of the playing pieces belongs to us, we set the variable *one_of_us* to TRUE. If one of the playing pieces belongs to any of the other players, we set the variable *one_of_them* to TRUE.

After the loops are finished, the two Boolean variables are checked. If one of our playing pieces is in the gold mine and no other player's piece is, then the code adds gold pieces to our cache of coins and the new value is displayed on the operations window.

Players with over Four Kills

When the war engine was first described, mention was made of players getting increased points when a playing piece that they are attacking is destroyed. Adding this code is fairly simple. In the OnWar() function, you can find the line of code

```
players[0]->kills++;
```

in several locations. This code is executed when a war outcome causes a playing piece to be destroyed. Each time this occurs, the *kills* variable is increased. We need to check the value of this variable to see if it is greater than 3. If the value is greater than 3, the current playing piece's offense points should be increased by a single point.

The code we are going to add to do the checking is

```
if (players[0]->kills > 3)
      {
        if (playing pieces[who_am_i][first_clicked]->offense < 11)
             playing_pieces[who_am_i][first_clicked]->offense++;

        players[0]->kills = 0;
        SendInfo("sprite_incr", who_am_i, first_clicked,
playing_pieces[who_am_i][first clicked]->offense, "");
      }
```

This code should be added just after the *switch* statement in the OnWar() function. The code simply checks to see if the *kills* variable is greater than 3. If this is true, an additional check is made to see if the current playing piece's offense points are 11 or greater. If the offense points are 11 or greater, we will not increase the offense points. The reason for this is that the war engine matrix is only indexes from 0 to 11. You can increase the size of the matrix and change this value if you want.

After the points are increased, the *kills* variable of the playing piece is reset and a packet is sent to all of the other players letting them know the new offense point value. This means we will need to have code in the OnCommand() function to handle the packet with the command SPRITE_INCR. The code to add to the OnCommand() function is

```
else if (strcmp(command, "sprite_incr") == 0)
   {
    playing_pieces[player2][sprite1]->offense = sprite2;
   }
```

The code just sets the playing piece's offense variable to the offense points value sent in the packet.

Castle Points

In addition to the kill points, all of the playing pieces will be able to take advantage of double defense points when they are in their own corners. The playing field has castles in each corner, which represent the home bases for each player. When a playing piece is attacked in its own home, it is given double the defensive points it currently owns.

The code for the defense points follows and should be placed in the OnWar() function just after the variables *offense* and *defense* are assigned values.

```
switch(second_clicked_player)
      {
       case 0: rect = new CRect(0,0,70,70);
               pnt =
CPoint(playing_pieces[second_clicked_player][second_clicked_icon]->mX,
playing_pieces[second_clicked_player][second_clicked_icon]->mY);

               if (rect->PtInRect(pnt))
                  defense *= 2;
               break;

       case 1: rect = new CRect(380,0,480,70);
               pnt =
CPoint(playing_pieces[second_clicked_player][second_clicked_icon]->mX,
playing_pieces[second_clicked_player][second_clicked_icon]->mY);

               if (rect->PtInRect(pnt))
                  defense *= 2;
               break;

       case 2: rect = new CRect(0,320,70,480);
               pnt =
CPoint(playing_pieces[second_clicked_player][second_clicked_icon]->mX,
playing_pieces[second_clicked_player][second_clicked_icon]->mY);

               if (rect->PtInRect(pnt))
                  defense *= 2;
               break;

       case 3: rect = new CRect(380,320,480,480);
               pnt =
CPoint(playing_pieces[second_clicked_player][second_clicked_icon]->mX,
playing_pieces[second_clicked_player][second_clicked_icon]->mY);

               if (rect->PtInRect(pnt))
                  defense *= 2;
               break;
    }
```

The code is a *switch* statement based on the attacked piece's player number. We will look at the code for player 0 as an example. If some player is attacking a playing piece that belongs to player 0, this code will follow the code in the *case 0* block. A rectangle is made of a portion of the corner in the upper left. The upper-left corner pixel of the playing piece is checked to see if the playing piece in within the rectangle created. If the playing piece is in the rectangle, the variable *defense* is doubled. Just above this code, we assigned the variable *defense* equal to the defense points of the attacked piece.

Summary

That's all of the details we are going to put into the game for this book. You have before you an example of a multiplayer sprite-based game. Expand on the game using your imagination. In the next chapter, we will add support for those using a modem for multiplayer gaming.

Al
ea
us
m
Th

W

Si
wi
pr

Ta

R
O
G
S
V

Chapter 11

Modem Support

Although the Internet has experienced tremendous growth for the past several years, not everyone has or even wants access. Many computer gamers use their modems to call a friend directly and play against one person at a time, so any multiplayer game you create should include modem support. This chapter will detail the addition of modem support for our game.

Windows Support for Modems

Since we don't have the luxury of this new library, we have to communicate with our modem using several COMM routines that Windows and its SDK provide for us. These routines are listed in Table 11.1.

Table 11.1 COMM Routines in the Windows SDK

Routine	*Explanation*
OpenComm()	Opens a COM port for activity
GetCommState()	Retrieves the current state of a COM port
SetCommState()	Sets the current state of a COM port
WriteComm()	Writes a string to a COM port

ReadComm()	Reads a string from a COM port
FlushComm()	Removes all characters from a COM port buffer
EnableComm Notification()	Registers the COM port with Windows in order to activate messages during certain COM operations
DCB	A structure used to hold information about a communication device

Initializing the Modem

When the players in the game want to connect with each other, they must select the appropriate network option. We currently only have one network: the Internet. All of the network options were set up on the menu when we designed the initialization of the Winsock library. We also added the code for the command update routine, which puts a check mark next to the selected option. All we need to do is to initialize the modem code when the modem option is first clicked.

The code to initialize the modem follows and must be put in the function OnNetworkSerialmodem().

```
COMMINPUT pDlg;
char buffer[30];
DCB dcb;

pDlg.current_baud = serial_baud_rate = 0;
pDlg.current_com = serial_com_port = 0;

// get the baud rate and com port to use
while (serial_baud_rate == 0 && serial_com_port == 0)
{
   pDlg.DoModal();

     serial_baud_rate = pDlg.current_baud;
     serial_com_port = pDlg.current_com;
     strcpy(serial_initialization_string, pDlg.m_initialization_string);
}
    //check for the modem
    sprintf(buffer, "COM%d", serial_com_port);

    COMPortID = OpenComm(buffer, 4096, 2048);
    if (COMPortID  0)
     MessageBox("Cannot open port", "ERROR", MB_OK);
    else
{
     //create communicaton DCB
     GetCommState(COMPortID, &dcb);
     dcb.BaudRate = serial_baud rate * 100;
     dcb.ByteSize = (BYTE)8;
     dcb.Parity = NOPARITY;
     dcb.StopBits = ONESTOPBIT;
     dcb.DsrTimeout = 1000;
     dcb.fDtrDisable = FALSE;

     if (SetCommState(&dcb) != 0)
      MessageBox("Unable to create comm structures", "ERROR", MB_OK);
     else
     {
      MessageBox("Serial Port Initialized", "Ok", MB_OK
      current network = NETWORK_MODEM;
     }
}
```

You will need the following declarations in the view object class.

```
long serial_baud_rate;
int com_port;
```

The code begins by zeroing out the current Baud and COM ports for the modem. The code enters a loop based on these two variables. What we are going to do is display a dialog box that allows the user to enter the COM port of their modem and the baud rate it should communicate at. If the user does not enter values for these two options, the dialog box will continue to be displayed until the values are entered.

The dialog box needs to be created in AppStudio. It appears as illustrated in Figure 11.1. The dialog box ID is IDD_COMM_INPUT. There are five controls:

1. OK button
2. Cancel button
3. COM port group
4. BAUD rate group
5. Modem Initialization Edit control

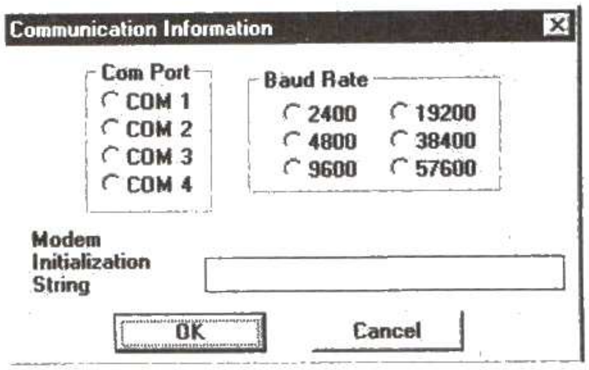


Figure 11.1 *The communication information dialog box.*

COM Port Group

The COM Port group includes four radio buttons. Each of the radio buttons is labeled for each of the four different COM ports. There are IDs for each button.

Label	*ID*
COM 1	IDC_COM_1
COM 2	IDC_COM_2
COM 3	IDC_COM_3
COM 4	IDC_COM_4

Using the group control, put all four of the radio buttons into a group and label it COM Port.

Baud Rate Group

The Baud Rate group includes six radio buttons. The radio buttons are defined as:

Label	*ID*
2400	IDC_BAUD_2400
4800	IDC_BAUD_4800

9600	IDC_BAUD_9600
19200	IDC_BAUD_19200
38400	IDC_BAUD_38400
57600	IDC_BAUD_57600

Using the group control, put all six of the radio buttons into a group and label it Baud Rate.

Modem Initialization Control

Not all modems are created alike. Some of them are high performance and others are plain-Jane. For this reason, some of them need special initialization strings to enable their special features. Not all of the modems need the initialization string, so the edit control will be blank and the player can add the necessary commands. The edit control should be given the ID name of IDC_INPUT_INITIALIZATION_STRING.

ClassWizard

Once the dialog box has been created using AppStudio, we need to move to the ClassWizard. The class for this new dialog box is called CommInput. In the Message Map area, create a message handler for all of the radio buttons, the OK button, and the Cancel button. Move to the Member Variable area and add a variable for the string initialization control. The name of the variable is *m_input_init_string*.

Class Code

With our dialog box designed, we have to add some code to make it useful. Open the file omminput.h and add the following declaration to the PUBLIC area of this class.

```
int current baud;
int current_com;
```

These variables will keep track of the different radio buttons the user clicks on. This means that when a specific radio button is clicked, code within the handler for the button must set the appropriate variable equal to the appropriate value. The code to do this is in the omminput.cpp file. Update the following functions to read as they do here.

```
void CommInput::OnBaud19200()
{
 current_baud = 192;
}

void CommInput::OnBaud2400()
{
 current baud = 24;
}

void CommInput::OnBaud38400()
{
 current_baud = 384;
}

void CommInput::OnBaud4800()
{
 current_baud = 48;
}

void CommInput::OnBaud57600()
{
 current_baud = 576;
}

void CommInput::OnBaud9600()
{
 current_baud = 96;
}

void CommInput::OnCom1()
{
 current_com = 1;
}

void CommInput::OnCom2()
{
 current_com = 2;
}

void CommInput::OnCom3()
{
 current_com = 3;
}

void CommInput::OnCom4()
{
 current_com = 4;
}

void CommInput::OnOK()
{
 CDialog::OnOK();
}
void CommInput::OnCancel()
{
 CDialog::OnCancel();
}
```

Every time the player clicks on a radio button, a member variable will be set equal to a value reflecting the state of the button.

Check for Modem

After the input dialog box has been filled with the necessary information, the code has to check for a modem. This is performed with

```
sprintf(buffer, "COM%d", serial_com_port);

 COMPortID = OpenComm(buffer, 4096, 2048);
 if (COMPortID  0)
  MessageBox("Cannot open port", "ERROR", MB_OK);
 else
```

The first line of code copies the text COM followed by the COM port of the modem. This is a required format for the COM port parameter of the Open-Comm() function, which is the second line of code. The second parameter is the receive buffer size and the third parameter is the transmit buffer for this COM port.

The result of the OpenComm() function is placed in a global variable called *COMPortID*. This variable will be used in both the WriteComm() and Read-Comm() functions when transmitting and receiving information between players. If by chance the code cannot find a modem, a message box is displayed warning the player.

Set Up Communications Information

Once the code has established that a modem is present, the code has to create a DCB structure with the necessary information about the serial port. If the serial port cannot be updated with the new information, a message is displayed to the user indicating the problem. If the serial port is updated successfully, a message box is displayed letting the player know that a connection can be tried.

Setting Up Players

At some point, players will have to enter information about the server in the game. They will use the same menu item and dialog box as for an Internet game. The only change comes when the client tries to connect with the server. The client will use the number entered as the server's Internet address/telephone number as a telephone number.

Creating a Connection

Obviously only two players are allowed to play a game via a modem connection. This means that the server will have to be ready for a client just as in the Internet case, but will only have to wait for one client. The same menu items and functions used for the Internet connection will be used for the modem connection as well. We will only be adding a single IF statement.

Server

Find the function OnWaitforconnect() and add the following code to the end of the file:

```
if (current_network == NETWORK_SERIAL)
 {
  if (EnableCommNotification(COMPortID, m_hWnd, 60, -1) == 0)
  {
   MessageBox("Comm notification failed", "Error", MB_OK);
   return;
  }
```

```
    //Flush both buffers
    FlushComm(COMPortID, 0);
    FlushComm(COMPortID, 1);

    //Modem Initialization String
    WriteComm(COMPortID, "AT S0=1\r", 8);
    WriteComm(COMPortID, serial_initialization_string,
strlen(serial_initialization_string));
    WriteComm(COMPortID, "\r", 1);
  }
```

This code will be executed only if the network we are using is for the modem. The next IF statement is one of the most important statements for the modem because it handles the registering of the modem communications with the Windows system. The function EnableCommNotification() tells Windows that we need to be notified by a message when any information comes through the modem. The message sent is WM_COMMNOTIFY. We will need to create a message handler for this message a little later. Once Windows knows about the communication through a serial port the code flushes both the transmit and receive buffers so that no unwanted characters will be transmitted to or received by the other player.

Next, we have to initialize the modem for communications and enable auto-answering. This process is performed in three stages. The first stage is to send the modem the auto-answering information. This information is the string *ATS0=1\r*. This string tells the modem to answer the modem on the first ring.

The next stage is to send the initialization string that the player might have entered to the modem. Notice that the initialization string must start with the command AT in order to be effectively received by the modem. The last stage is to send a newline to the modem. Everything is now ready for the server to receive a client.

Client

The client code that follows needs to be added to the function OnConnect-toserver().

```
if (current_network == NETWORK_SERIAL)
    {
      if (total_players > 2)
      {
       MessageBox("Sorry but you cannot have > 2 players on a serial connection",
"Error", MB_OK | MB_ICONSTOP);
      }
      else
      {
       //Flush both buffers
       FlushComm(COMPortID, 0);
       FlushComm(COMPortID, 1);

       if (EnableCommNotification(COMPortID, m_hWnd, 60, -1) == 0)
       {
        MessageBox("Comm notification failed", "Error", MB_OK);
        return;
       }

       if (strlen(the_server->address) > 0)
       {
        WriteComm(COMPortID, "ATDT ", 5);
        WriteComm(COMPortID, the_server->address, strlen(the_server->address));
        WriteComm(COMPortID, "\r", 1);
      }
   }
}
```

The code begins by making sure there are a total of two players set up to play the game. If there is any other value for the variable *total_players*, the code displays a warning and does not allow the client to attach to the server.

As in the case of the server, the buffers are flushed to rid any unwanted characters and the system is set up to send the WM_COMMNOTIFY message when there is communication through the modem. The last piece of code does the actual dialing of the modem. This is performed in three stages. First the command ATDT is sent to the modem to signal touch tone dialing. Next, the telephone number of the server is dialed, and last, a newline is sent to the modem.

CommNotify Handler

Now that each machine, client and server, has the code to connect with each other, we need to add the code which handles all of the communication from the modem. There needs to be a message macro that relates the message WM_COMMNOTIFY with a message handler. Add the following message map.

```
ON_MESSAGE(WM_COMMNOTIFY, OnCommNotify)
```

The handler itself is

```
long CNetwar2View::OnCommNotify(UINT wParam, LONG lParam)
{
   char buffer[65];
   int result;

   if (wParam == (UINT)COMPortID)
   {
    switch(LOWORD(lParam))
    {
     case CN_RECEIVE: result = ReadComm(COMPortID, buffer, 60);
                          if (result  0)
                             MessageBox("Serial Port Read Error", "Error",
MB OK);
                          else
                          {
                             buffer[result] = '\0';
                          DoCommand(buffer);
                      }
                      break;
   }
  }

  return 1;
}
```

This is the function that will be called when any characters are received from the modem. Both of the parameters to this function contain information that we need to look at. The first IF statement is used to check for characters coming from the modem the game registered. This IF statement is necessary because a program can register several modems and the same function will be called for all of them.

The second statement, *switch,* is used to check and see if the function was called because a character was received from the modem. Other modem situations can trigger this function, but we don't need to use any of them.

When the *lParam* variable is equal to the constant CN_RECEIVE, a packet has arrived at the modem. The code in the *case* statement reads the packet from the modem and puts it in the local variable *buffer.* A check is made of the return value from the ReadComm() function in order to respond to a possible error. If the reception of the packet is good, an end-of-line character is appended to the buffer and the packet is processed in the DoCommand() function.

This is all the code we need to receive packets from our opponent in the game. The only other routine needed is sending information.

Connection

When a connection occurs between the server and the client, the modem will send each of the players a connect message. This message has the text CONNECT in it. We will use this information to instruct the server to begin sending the client information about itself. In the DoCommand() function, add the following code.

```
else if (strncmp(command, "CONNECT", 7) == 0)
 {
  if (server == 0)
  {
   SendInfo("who", 1, 0, 0, "");
   SendInfo("game_ready", 0, 0, 0, "");

   MessageBox("All Connected. Begin Game", "Go!", MB_OK);

   gold_timer = SetTimer(1, 1000, NULL);

   war_button = 0;
   war_timer = SetTimer(2, 2000, NULL);

   ops_dlg ->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
  }
 }
```

This code should look familiar, as it is the same code as the Internet game executed when all players have connected with the server. Since we only have one client, we can execute the code when the modems have connected.

Sending Information

When any packet, such as a playing piece selection, needs to be sent to our opponent, we have to send the packet through the modem instead of using the Internet. This means that everywhere we put the SendInfo() function call for the Internet transmission, we need to place a call to a function to handle the modem transmission. We had better hope we place the new function in all of the correct places. In addition, we will have to have some sort of IF statement that will choose the correct transmission function based on the current network.

Actually, we are just going to leave the SendInfo() function calls and change the code in this function instead of adding code to all of the different transmission places in the code. The new SendInfo() function looks like

```
int i;

 sprintf(send_buffer, "%11s %3d %3d %3d %3d %32s", command, data1, data2, data3,
who_am_i, msg);

 if(server == 0)
 {
  if (current network == NETWORK_INTERNET)
  {
   for (i=1;itotal_players;i++)
    send(the_server->clients[i-1], send_buffer, 60, 0);
  }

  if (current_network == NETWORK_SERIAL)
   WriteComm(COMPortID, send_buffer, 60);
 }
 else
 {
  if (current_network == NETWORK_INTERNET)
   send(players[0]->socket, send buffer, 60, 0);

  if (current network == NETWORK_SERIAL)
   WriteComm(COMPortID, send buffer, 60);
 }
```

The only change to the function is the addition of the *WriteComm* statements. Instead of sending the packet using the Internet, the modem is used.

Acknowledgments

During game play over the modem, you might find that things get out of sync or packets are lost. This is due to the communication delays and poor quality of telephone lines. To help combat these problems, we can add acknowledgments to the game. Every time information is sent from one machine to another, the sender will wait for an acknowledgment. If an acknowledgment does not arrive in 3 seconds, the sender assumes the packet has been lost and resends it. The receiver of the packet must send an acknowledgment packet back to the sender as soon as it receives a packet.

Just as adding an extra invalidate was an option, acknowledgments will be as well. Create a menu item under Options called Acknowledgements. Create command and update command functions for the menu item in ClassWizard.

The two new functions should be updated as

```
void CNetwar2View::OnOptionsAcknowledgements()
{
 if (ACTIVATE_ACKS)
  ACTIVATE_ACKS = FALSE;
 else
  ACTIVATE_ACKS = TRUE;
}
void CNetwar2View::OnUpdateOptionsAcknowledgements(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(ACTIVATE_ACKS == TRUE);
}
```

The variable *ACTIVATE_ACKS* will need to be declared as a Boolean variable in the view object's class. If this variable has a value of TRUE, then acknowledgments should be used in the game. This means that every time the function SendInfo() is executed, the machine should wait for an acknowledgment back. In the SendInfo() function, add the following lines of code.

```
if (ACTIVATE_ACKS)
 get_ack();
The function get_ack() is
void CNetwar2View::get_ack()
{
 MSG msg;
 BOOL SUCCESS;

 SUCCESS = FALSE;
 time_up = FALSE;
 total_acks = 0;
 while (!SUCCESS)
 {
  send_timer = SetTimer(5, 3000, NULL);
  if (current_network != NETWORK_NONE
  {
   while ((total_acks  (total_players-1)) && (GetMessage(&msg, m_hWnd, 0, 0)))
   {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
   }
```

```
      KillTimer(send_timer);
      if ((time_up == TRUE) && (total_acks  (total_players-1)))
      {
        total_acks = 0;
        if(server == 0)
          {
            if (current_network == NETWORK_MODEM)
              WriteComm(COMPortID, send_buffer, 60);
          }
          else
          {
            if (current_network == NETWORK_MODEM)
              WriteComm(COMPortID, send_buffer, 60);
          }
        }
        else
        {
          SUCCESS = TRUE;
        }
      }
    }
}
```

The function works like this: A variable called *total_acks* is set to 0. A loop is entered that simply processes messages. When a player acknowledges the sent packet, the variable *total_acks* is incremented. When all of the players have acknowledged the packet, the code continues. Notice that the code allows for acknowledgments over the Internet as well.

At the same time as all of this, a timer has been started with an expiration time of 3 seconds. If all of the acknowledgments have not been received in this time frame, the packet is resent. For all of this to work, we need to have a command handler in the DoCommand() function for the acknowledgment packet. The necessary code is

```
else if (strcmp(command, "ack") == 0)
   {
     if (player2 == who_am_i)
     {
       total_acks++;
     }
   }
```

The code checks to make sure that the acknowledgment packet is meant for this player and increments the *total_acks* variable.

Now what about the machine that receives the packet and needs to send an acknowledgment packet? This machine will send the acknowledgment just before it tries to process the packet. Add the following code at the top of the DoCommand() function.

```
if (ACTIVATE_ACKS)
 send_ack(player1);
```

The function send_ack() is

```
void CNetwar2View::send_ack(int from)
{
 char buffer[70];


 if(server == 0)
 {
  sprintf(buffer, “%11s %3d %3d %3d %3d %32s”, “ack”, from, 0, 0, 0, “”);

  if (current_network == NETWORK_TCP)
  {
   for (i=1;itotal_players;i++)
    send(the_server->clients[i-1], send_buffer, 60, 0);
  }

  if (current_network == NETWORK_MODEM)
   WriteComm(COMPortID, buffer, 60);
  }
  else
  {
   sprintf(buffer, “%11s %3d %3d %3d %3d %32s”, “ack”, from, 0, 0, who_am_i, “”);

   if (current_network == NETWORK_TCP)
    send(players[0]->socket, buffer, 60, 0);

   if (current_network == NETWORK_MODEM)
    WriteComm(COMPortID, buffer, 60);
  }
}
```

You will notice the code looks the same as the SendInfo() function except there is no IF statement at the bottom of the function to receive acknowledgment for this packet. We don't want to use the SendInfo() function because we would be requesting acknowledgment on acknowledgment packets. This would cause a never-ending downward spiral into computer nothingness or a crash.

Before all of this code will work, we have to add code to the OnTimer() function. The code to add is

```
if (nIDEvent == (UINT)send_timer)
 {
  time_up = TRUE;
 }
```

When this timer expires, the variable *time_up* is set to TRUE and a duplicate packet will be sent to the players. One last piece of code needs to appear in the view constructor:

```
ACTIVATE_ACKS = FALSE;
```

All of the games will start with no acknowledgments. They take time and resources, and should not be used unless necessary.

Summary

Playing the game over the modem can be just as exciting as using the Internet. Although you lose the multiplayer aspect by only having a single opponent, you could extend the code for additional modems and telephone lines if you have the available hardware.

Chapter 12

EXPANSION

Chapters 1 to 11 guided you through the process of creating a complete multiplayer game that works over modems, Ethernet networks, and the Internet. The game can be played between two to four players and includes purchasing equipment, moving units around the board, fighting battles, sending messages back and forth among players, and more. There are always a few more features you can add—here are a few suggestions.

Windows for Workgroups

During the sequence of Microsoft Windows environments, Windows for Workgroups was introduced to fill the need for peer-to-peer networking. Users of WfW can share information between themselves such as drives and printers. This does not seem like something we can use for our game, but as you look through the Microsoft documentation, you will find two things that will allow our game to be expanded to use WfW.

Before we move into the details, the reason adding WfW support might be the fact that it does not require the Internet. The WfW computers are attached via standard ethernet thin-line cable. Not requiring the Internet gives an added bonus to your network games.

One of the ways we can use WfW to network our game is through Network Dynamic Data Exchange. NDDE is a form of memory shared between processors connected with WfW. Most of the functionality of NDDE is handled by Windows itself and is started automatically when Windows is started.

The second way of communicating information from one computer to another is through a mailslot. A mailslot is just a location on a machine that a string of information can be put. Creating a mailslot is as easy as using the function DosMakeMailslot(). Putting information into the mailslot uses the function DosWriteMailslot(). You can write information into any mailslot created on a machine.

The reason you do not see more code in this area is because a developer is required to be part of the Microsoft Developers Network at Level II in order to have access to the libraries necessary for compiling code. This is a $500 commitment.

Dynamic IP

Many people who are connecting to the Internet are doing it through an Internet provider. This connection is through either a SLIP or PPP connection. When people connect to their provider using one of these protocols, they are dynamically assigned an IP address. This isn't a problem for the game, since all we need is a single server with an IP. The clients don't all need to know their IP address.

Additional Playing Pieces

For space reasons in the book, only four playing pieces were designed to be purchased and used in the game. A game that you design will probably have more pieces. In addition to the playing piece count, you could add more characteristics to the pieces, including such things as health, armor, and others.

Animated Playing Pieces

You can add animation to the playing pieces as well, if you wish. A tank playing piece could actually move over the playing field. The playing field could have animation in it as well. Imagine a mountain where an occasional avalanche occurs. The avalanche would take out any playing pieces located on the mountain.

More Players

A server in the game can handle up to 20 socket connections. Handling these connections will require a powerful computer, but would provide a very powerful game. At this point, you might want to keep the server as only a server and not a player in the game as we have things. This would give the server the only responsibility of routing messages from player to player.

Music and Sound Effects

The last addition we will talk about is music and sound. Background music would be very important and interesting in the game—some kind of soft and low music that would allow sound effects to be clearly heard over the background. Sound effects could be added for movements, wars, and the various outcomes.

Appendix A

Playing King's Reign

The King's Reign game requires a minimum of two players and allows up to four players to compete simultaneously. The game supports modems and the Internet (Winsock). There is also a built-in provision to allow a single player to investigate some of the features of the game.

The players in the game are designated as either servers or clients. There can only be one server in the game. The person playing at the server machine will need to know his or her own Internet IP address or telephone number. This information needs to be communicated to all of the other players (clients) in the game.

Starting the Game

The game is started by executing the kingsm.exe. file in the /kingsr directory on the enclosed CD-ROM. Upon execution, you will receive two windows on your screen like the ones shown in Figure A.1.

The large window is the playing field for the game. The smaller window is an operations window and will contain information about you and the game. At this point, you and all of the players will need to enter the total number of players in the game and server information.

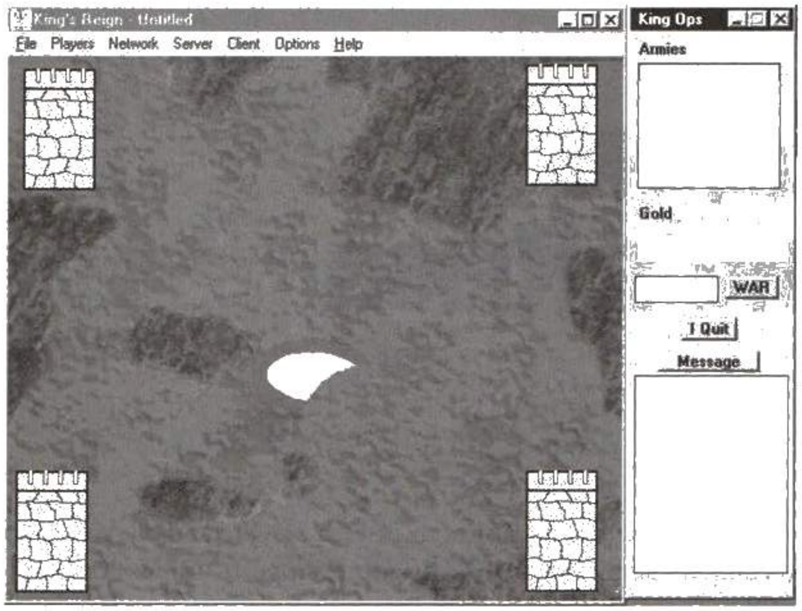


Figure A.1 *Starting King's Reign.*

Click on the menu item called Players followed by a click on the entry for Input Players. You will receive a dialog box like the one shown in Figure A.2

Click on one of the two radio buttons. Only one machine will click on the Server option. The remainder of the players will click on Client. Now click in the edit control labeled for the total number of players in the game and enter

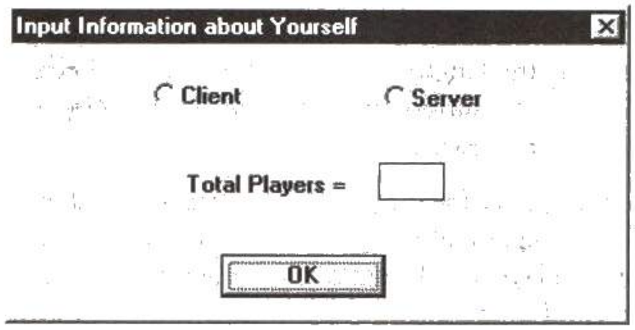


Figure A.2 *The Input Players dialog box.*

the appropriate value. Click on OK to continue entering player information.

The player that clicked on the Server radio button will not receive a second dialog box. All of the players that clicked on Client will receive the dialog box shown in Figure A.3.

All of the clients need to enter the server's Internet IP or telephone number in the edit control on this dialog box. After this has been done, click on the OK button.

Connecting to the Server

Once the player information is entered into the game, the clients can connect to the server. First all of the players must choose the network that is being used. The only two options available are Internet (Winsock) and serial (modem). Click on the menu option Network followed by a click on the appropriate network to use.

One of two things will occur. If you click on Winsock, a dialog box will appear as shown in Figure A.4 if the Winsock DLL was found and loaded correctly on your system.

Click on OK to continue. If you click on Serial, you will receive the dialog box shown in Figure A.5.

This dialog box allows you to enter information about your modem. Click on the appropriate Com Port and Baud Rate to use for your modem. You can

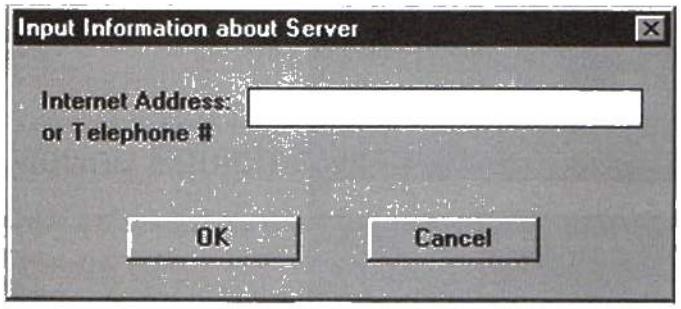
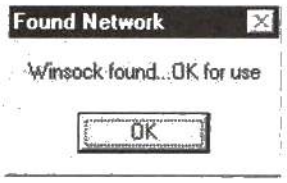


Figure A.3 *Server/Client Player Input dialog box.*


Found Network

Winsock found...OK for use

Figure A.4 *Winsock Error dialog box.*

enter an initialization string in the edit control on the dialog box. The initialization string *must* start with *at* to be effective. Click on OK when finished. The system will try to access your modem and let you know the outcome through a message box.

We are now ready to connect the machines. The player who is the server must click on the menu option Wait for Connect before the clients try to connect. After clicking on this menu item, the dialog box shown in Figure A.6 will appear. The server is now waiting for clients to connect.

All of the clients will now click on the menu item Connect to Server. The system will try to connect to the server. When successful, a dialog box will appear letting the players know their player numbers in the game. Once all of the players are connected, a dialog box will appear on all of the machines, letting the players know the game is ready to be played. Click on OK to start game play.

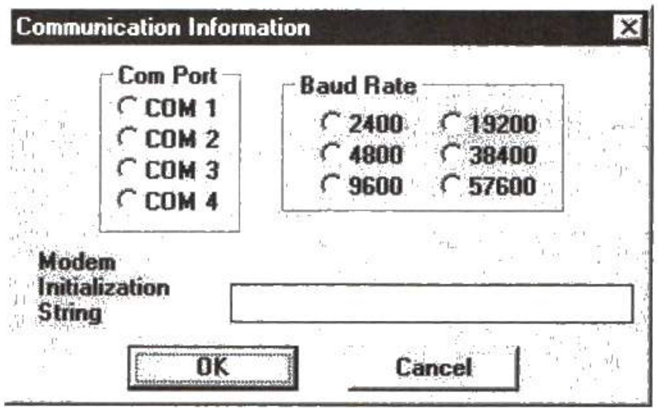
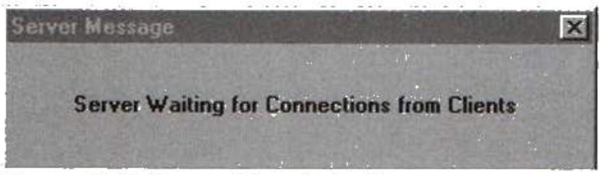


Figure A.5 *Serial Port Setup dialog box.*


Server Message

Server Waiting for Connections from Clients

Figure A.6 *Server Wait dialog box.*

Buying Equipment

The first thing the players will want to do after connecting is to purchase playing pieces. Click on Players and then click on Buy Equipment. You will get the dialog box shown in Figure A.7.

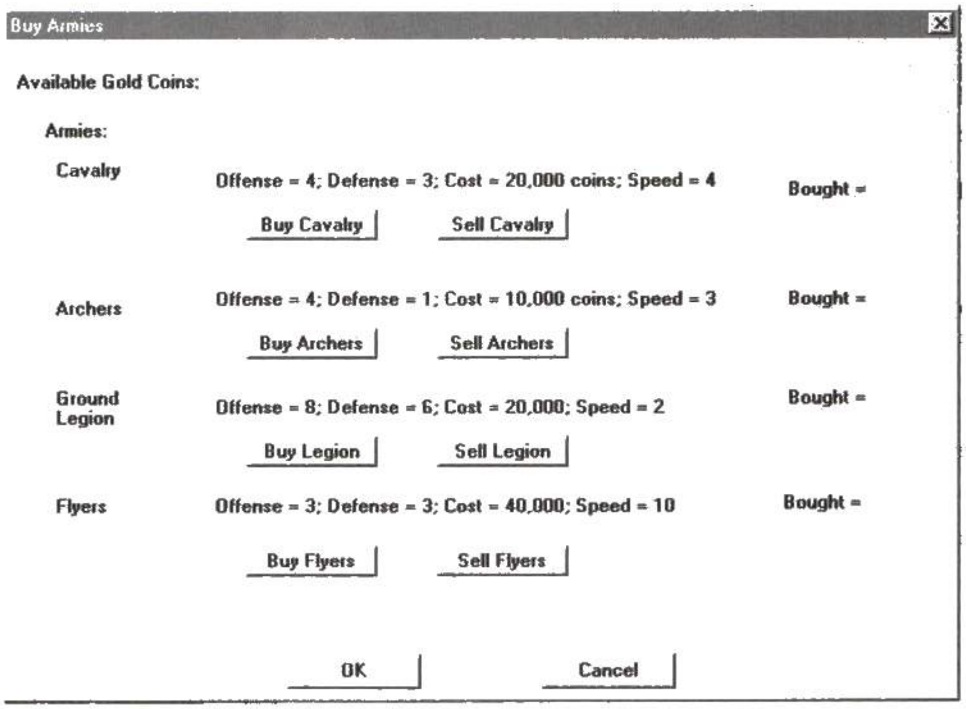
This dialog box allows you to purchase playing pieces. During a purchase, you can buy and sell playing pieces depending on your gold pieces. Each player starts the game with 500,000 gold coins. Once you have purchased your playing pieces, click on OK. All of the playing pieces purchased will be listed in the Armies list box on the operations window. This is shown in Figure A.8.

Placing Equipment

Armies listed in the Armies list box on the operations window can be placed on the playing field. To do this, double-click on one of the names in the list box. The name of the playing piece will be changed to Army Deployed, and a playing piece will appear on the playing field as shown in Figure A.9. All the players will have their playing pieces placed in one of the four corners. This is the castle or home position.

Moving Equipment

Once a playing piece is on the playing field it can be moved. To move a playing piece, move the mouse so that the arrow is on one of your own playing pieces. Now click and hold the left mouse button. Move the mouse to move the playing piece. Release the left mouse button to stop moving the playing piece.


Buy Armies

Available Gold Coins:

Armies:

Cavalry	Offense = 4; Defense = 3; Cost = 20,000 coins; Speed = 4	Bought =
Archers	Offense = 4; Defense = 1; Cost = 10,000 coins; Speed = 3	Bought =
Ground Legion	Offense = 8; Defense = 6; Cost = 20,000; Speed = 2	Bought =
Flyers	Offense = 3; Defense = 3; Cost = 40,000; Speed = 10	Bought =

Figure A.7 *The Buy Equipment dialog box.*

All of the playing pieces have a limiting factor. The Flyer playing piece can move the largest distance per left mouse button click. The Legion playing piece is the least movable.

Playing pieces can be moved anywhere on the playing field, but you can only move your own pieces.

Starting a War

After moving your playing pieces into position, you will want to start a war. To do this, you have to wait for a war token to be at your machine. You know when this has occurred because the text Okay For will be displayed to the left of the WAR button on the operations window.

When this text is on your operations window, double-click on the playing piece you want to use for a war. A yellow highlight will appear on your piece.

Now double-click on the opponent's playing piece. It will also be highlighted. Click on the WAR button on the operations window. If you are close enough to the opponent, a war will take place. The system will display who is having the war and its outcome in the message list box on the operations window.

If the opponent is destroyed, it will be removed from the playing field. After your war, both playing pieces will be unhighlighted and the war token will be sent to another player.

Using the Gold Mine

In the middle of the playing field is a gold mine. This gold mine dispenses 10 gold coins per second to the playing piece in the mine. The only catch is that only one playing piece is allowed in the mine at a time. If two or more pieces from opposing armies are in the mine, no gold coins are dispensed.

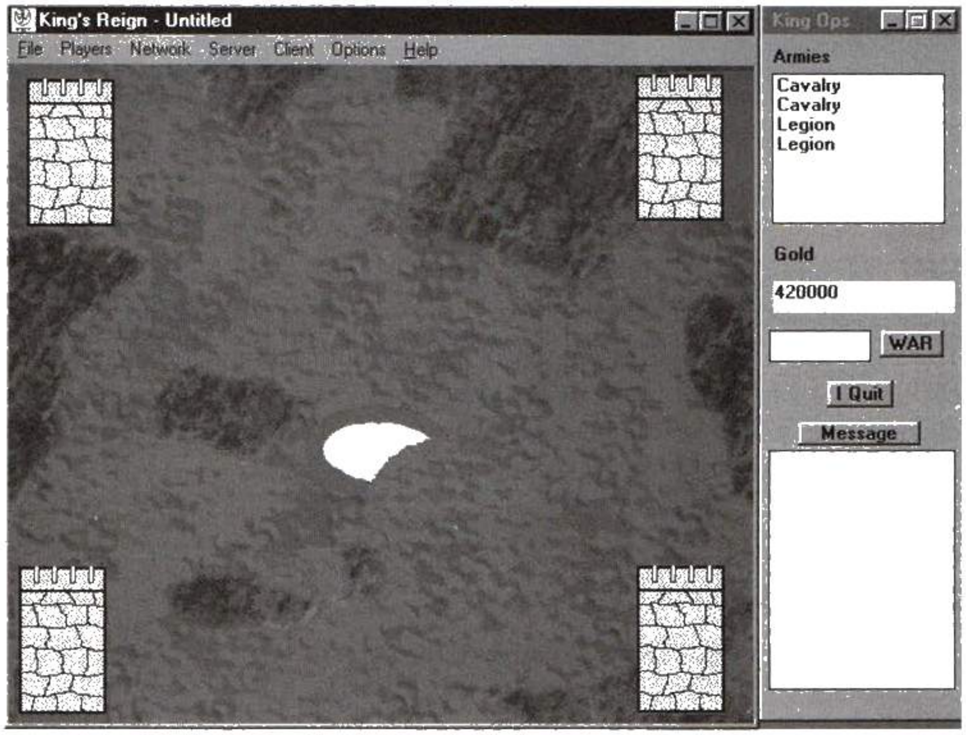


Figure A.8 *Armies list box on operations window.*

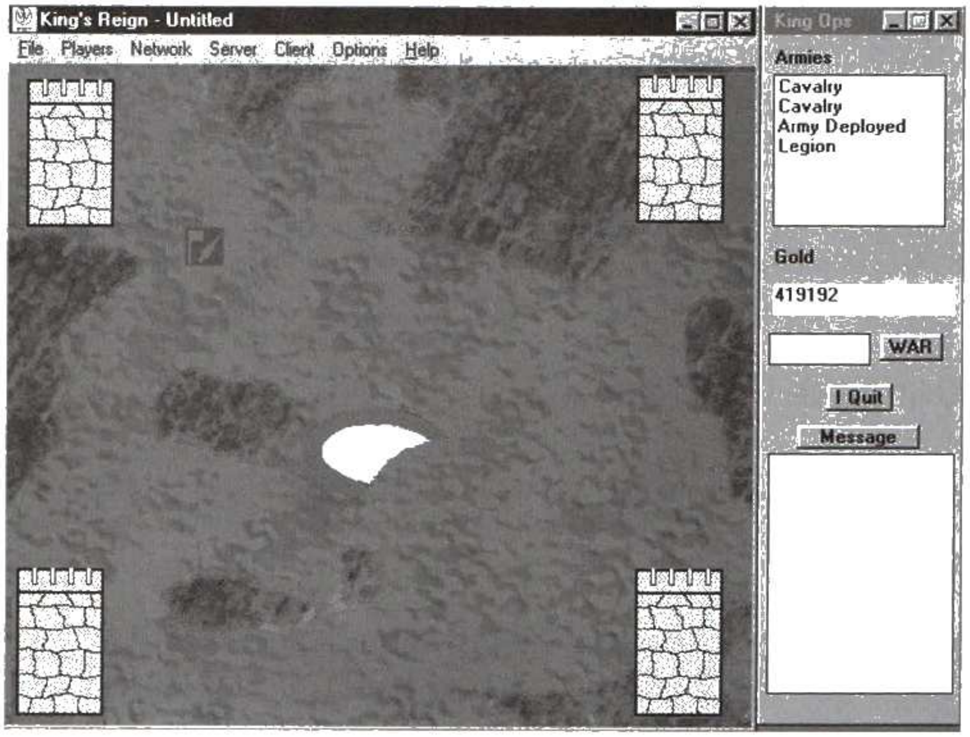


Figure A.9 *Army deployed to playing field.*

Sending a Message

You can send a message to one or all players during the game. To do this, click on the Message button in the operations window. You will receive a dialog box in which you can enter the message and click on a button to send the message to the appropriate player(s).

Special Features

A couple of extra features are available to the players.

Acknowledgments

During modem play, you might want to activate acknowledgments to cut down on problems with lost information between players. Click on the Acknowledgments entry under the Options menu item *before* connecting to the server. The server must also click on this option.

Extra Invalidates

You can help clear your playing field when all four players are playing the game by clicking on the Extra Invalidate menu item under the Options menu item. This will cause the window to be cleared each time a playing piece is moved.

Sound

There is some sound in the game. To activate the sound, click on the Sound menu entry under the Options menu item. You will have to have registered a sound device with Windows in order to hear the sound.

Ending the Game

To end the game, just exit or close the game's main window. This will automatically hang up the modem or close your Internet connection.

Appendix B

Complete Code Listing for King's Reign (Multiplayer Version)

buy_equip.h

```
// buy equi.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// buy equipment dialog

class buy_equipment : public CDialog
{
// Construction
public:
	buy_equipment(CWnd* pParent = NULL); // standard constructor

	long total_gold_coins;
	int archers_bought,
	    cavalrys_bought,
	    flyers_bought,
	    legions_bought;
// Dialog Data
	//{{AFX_DATA(buy_equipment)
	enum { IDD = IDD_BUY_EQUIPMENT };
	CString	m_buy_archers_count;
	CString	m_buy_available_coins;
	CString	m_buy_cavalrys_count;
	CString	m_buy_flyers_count;
	CString	m_buy_legions_count;
	//}}AFX_DATA

// Implementation
protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV support

	// Generated message map functions
	//{{AFX_MSG(buy_equipment)
	afx_msg void OnBuyArchers();
	afx_msg void OnBuyCavalrys();
	afx_msg void OnBuyFlyers();
	afx_msg void OnBuyLegion();
	afx_msg void OnSellArchers();
	afx_msg void OnSellCavalrys();
	afx_msg void OnSellFlyers();
	afx_msg void OnSellLegion();
	virtual void OnCancel();
	virtual void OnOK();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

buy_equip.cpp

```
// buy_equi.cpp : implementation file
//

#include "stdafx.h"
#include "netwar2.h"
#include "buy_equi.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// buy_equipment dialog
```

```
buy_equipment::buy_equipment(CWnd* pParent /*=NULL*/)
  : CDialog(buy_equipment::IDD, pParent)
{
        //{{AFX_DATA_INIT(buy_equipment)
        m_buy_archers_count = "";
        m_buy_available_coins = "";
        m_buy_cavalrys_count = "";
        m_buy_flyers_count = "";
        m_buy_legions_count = "";
        //}}AFX_DATA_INIT

        cavalrys_bought = 0;
        archers_bought = 0;
        flyers_bought = 0;
        legions_bought = 0;
}

void buy_equipment::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(buy_equipment)
        DDX_Text(pDX, IDC_BUY_ARCHERS_COUNT, m_buy_archers_count);
        DDX_Text(pDX, IDC_BUY_AVAILABLE_COINS, m_buy_available_coins);
        DDX_Text(pDX, IDC_BUY_CAVALRYS_COUNT, m_buy_cavalrys_count);
        DDX_Text(pDX, IDC_BUY_FLYERS_COUNT, m_buy_flyers_count);
        DDX_Text(pDX, IDC_BUY_LEGIONS_COUNT, m_buy_legions_count);
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(buy_equipment, CDialog)
        //{{AFX_MSG_MAP(buy_equipment)
        ON_BN_CLICKED(IDC_BUY_ARCHERS, OnBuyArchers)
        ON_BN_CLICKED(IDC_BUY_CAVALRYS, OnBuyCavalrys)
        ON_BN_CLICKED(IDC_BUY_FLYERS, OnBuyFlyers)
        ON_BN_CLICKED(IDC_BUY_LEGION, OnBuyLegion)
        ON_BN_CLICKED(IDC_SELL_ARCHERS, OnSellArchers)
        ON_BN_CLICKED(IDC_SELL_CAVALRYS, OnSellCavalrys)
        ON_BN_CLICKED(IDC_SELL_FLYERS, OnSellFlyers)
        ON_BN_CLICKED(IDC_SELL_LEGION, OnSellLegion)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// buy_equipment message handlers

void buy_equipment::OnBuyArchers()
{
 CEdit* temp;
 char buffer[12];
```

```
buy_equipment::buy_equipment(CWnd* pParent /*=NULL*/)
  : CDialog(buy_equipment::IDD, pParent)
{
	//{{AFX_DATA_INIT(buy_equipment)
	m_buy_archers_count = "";
	m_buy_available_coins = "";
	m_buy_cavalrys_count = "";
	m_buy_flyers_count = "";
	m_buy_legions_count = "";
	//}}AFX_DATA_INIT

	cavalrys_bought = 0;
	archers_bought = 0;
	flyers_bought = 0;
	legions_bought = 0;
}

void buy_equipment::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(buy_equipment)
	DDX_Text(pDX, IDC_BUY_ARCHERS_COUNT, m_buy_archers_count);
	DDX_Text(pDX, IDC_BUY_AVAILABLE_COINS, m_buy_available_coins);
	DDX_Text(pDX, IDC_BUY_CAVALRYS_COUNT, m_buy_cavalrys_count);
	DDX_Text(pDX, IDC_BUY_FLYERS_COUNT, m_buy_flyers_count);
	DDX_Text(pDX, IDC_BUY_LEGIONS_COUNT, m_buy_legions_count);
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(buy_equipment, CDialog)
	//{{AFX_MSG_MAP(buy_equipment)
	ON_BN_CLICKED(IDC_BUY_ARCHERS, OnBuyArchers)
	ON_BN_CLICKED(IDC_BUY_CAVALRYS, OnBuyCavalrys)
	ON_BN_CLICKED(IDC_BUY_FLYERS, OnBuyFlyers)
	ON_BN_CLICKED(IDC_BUY_LEGION, OnBuyLegion)
	ON_BN_CLICKED(IDC_SELL_ARCHERS, OnSellArchers)
	ON_BN_CLICKED(IDC_SELL_CAVALRYS, OnSellCavalrys)
	ON_BN_CLICKED(IDC_SELL_FLYERS, OnSellFlyers)
	ON_BN_CLICKED(IDC_SELL_LEGION, OnSellLegion)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// buy_equipment message handlers

void buy_equipment::OnBuyArchers()
{
 CEdit* temp;
 char buffer[12];
 if (total_gold_coins-10000 > 0)
 {
  archers_bought++;
  total_gold_coins -= 10000;

  temp = (CEdit*)GetDlgItem(IDC_BUY_ARCHERS_COUNT);
  sprintf(buffer, "%d", archers_bought);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);

  temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
  sprintf(buffer, "%ld", total_gold_coins);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);
 }
 else
 {
  MessageBox("You are out of money", "No More", MB_OK);
 }
}

void buy_equipment::OnBuyCavalrys()
{

 CEdit* temp;
 char buffer[12];

 if (total_gold_coins-20000 > 0)
 {
  cavalrys_bought++;
  total_gold_coins -= 20000;

  temp = (CEdit*)GetDlgItem(IDC_BUY_CAVALRYS_COUNT);
  sprintf(buffer, "%d", cavalrys_bought);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);

  temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
  sprintf(buffer, "%ld", total_gold_coins);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);

 }
 else
 {
  MessageBox("You are out of money", "No More", MB_OK);
 }
}

void buy_equipment::OnCancel()
{
      CDialog::OnCancel();
}


void buy_equipment::OnSellArchers()
{
  CEdit* temp;
  char buffer[12];

  if (archers_bought == 0)
    MessageBox("You have not bought any archers", "Cannot Sell", MB_OK);
  else
  {
    archers_bought--;
    total_gold_coins += 10000;

    temp = (CEdit*)GetDlgItem(IDC_BUY_ARCHERS_COUNT);
    sprintf(buffer, "%d", archers_bought);
    temp->SetSel(0, -1, FALSE);
    temp->ReplaceSel(buffer);

    temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
    sprintf(buffer, "%ld", total_gold_coins);
    temp->SetSel(0, -1, FALSE);
    temp->ReplaceSel(buffer);

  }

}

void buy_equipment::OnSellCavalrys()
{
  CEdit* temp;
  char buffer[12];

  if (cavalrys_bought == 0)
    MessageBox("You have not bought any cavalrys", "Cannot Sell", MB_OK);
  else
  {
    cavalrys_bought--;
    total_gold_coins += 20000;

    temp = (CEdit*)GetDlgItem(IDC_BUY_CAVALRYS_COUNT);
    sprintf(buffer, "%d", cavalrys_bought);
    temp->SetSel(0, -1, FALSE);
    temp->ReplaceSel(buffer);

    temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
    sprintf(buffer, "%ld", total_gold_coins);
    temp->SetSel(0, -1, FALSE);
    temp->ReplaceSel(buffer);
```

```
   }

}

void buy_equipment::OnBuyFlyers()
{
 CEdit* temp;
 char buffer[12];

 if (total_gold_coins-40000 > 0)
 {
  flyers_bought++;
  total_gold_coins -= 40000;

  temp = (CEdit*)GetDlgItem(IDC_BUY_FLYERS_COUNT);
  sprintf(buffer, "%d", flyers_bought);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);

  temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
  sprintf(buffer, "%ld", total_gold_coins);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);
 }
 else
 {
  MessageBox("You are out of money", "No More", MB_OK);
 }
}

void buy_equipment::OnBuyLegion()
{
 CEdit* temp;
 char buffer[12];

 if (total_gold_coins-20000 > 0)
 {
  legions_bought++;
  total_gold_coins -= 20000;

  temp = (CEdit*)GetDlgItem(IDC_BUY_LEGIONS_COUNT);
  sprintf(buffer, "%d", legions_bought);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);

  temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
  sprintf(buffer, "%ld", total_gold_coins);
  temp->SetSel(0, -1, FALSE);
  temp->ReplaceSel(buffer);
 }
 else
```

```
  {
   MessageBox("You are out of money", "No More", MB_OK);
  }
 }

void buy_equipment::OnSellFlyers()
{
      CEdit* temp;
  char buffer[12];

      if (flyers_bought == 0)
       MessageBox("You have not bought any flyers", "Cannot Sell", MB_OK);
      else
      {
       flyers_bought--;
       total_gold_coins += 40000;

      temp = (CEdit*)GetDlgItem(IDC_BUY_FLYERS_COUNT);
      sprintf(buffer, "%d", flyers_bought);
      temp->SetSel(0, -1, FALSE);
      temp->ReplaceSel(buffer);

      temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
      sprintf(buffer, "%ld", total_gold_coins);
      temp->SetSel(0, -1, FALSE);
      temp->ReplaceSel(buffer);
     }
}

void buy_equipment::OnSellLegion()
{
       CEdit* temp;
  char buffer[12];

       if (legions_bought == 0)
        MessageBox("You have not bought any Legions", "Cannot Sell", MB_OK);
       else
       {
        legions_bought--;
        total_gold_coins += 20000;

      temp = (CEdit*)GetDlgItem(IDC_BUY_LEGIONS_COUNT);
      sprintf(buffer, "%d", legions_bought);
      temp->SetSel(0, -1, FALSE);
      temp->ReplaceSel(buffer);

      temp = (CEdit*)GetDlgItem(IDC_BUY_AVAILABLE_COINS);
      sprintf(buffer, "%ld", total_gold_coins);
      temp->SetSel(0, -1, FALSE);
      temp->ReplaceSel(buffer);
   }
```

```
}

void buy_equipment::OnOK()
{
     CDialog::OnOK();
}
```

inputsel.h

```
 // inputsel.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CInputSelf dialog

class CInputSelf : public CDialog
{
// Construction
public:
       CInputSelf(CWnd* pParent = NULL); // standard constructor
       int type;

// Dialog Data
       //{{AFX_DATA(CInputSelf)
       enum { IDD = IDD_INPUT_SELF };
       CString  m_total_players;
       //}}AFX_DATA

// Implementation
protected:
       virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

       // Generated message map functions
       //{{AFX_MSG(CInputSelf)
       afx_msg void OnRadioClient();
       afx_msg void OnRadioServer();
       afx_msg void OnOk();
       //}}AFX_MSG
       DECLARE_MESSAGE_MAP()
};
```

inputsel.cpp

```
// inputsel.cpp : implementation file
//
```

```
#include "stdafx.h"
#include "netwar2.h"
#include "inputsel.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CInputSelf dialog


CInputSelf::CInputSelf(CWnd* pParent /*=NULL*/)
      : CDialog(CInputSelf::IDD, pParent)
{
      //{{AFX_DATA_INIT(CInputSelf)
      m_total_players = "";
      //}}AFX_DATA_INIT
}

void CInputSelf::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CInputSelf)
      DDX_Text(pDX, IDC_INPUT_TOTAL_PLAYERS, m_total_players);
      //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CInputSelf, CDialog)
      //{{AFX_MSG_MAP(CInputSelf)
      ON_BN_CLICKED(IDC_RADIO_CLIENT, OnRadioClient)
      ON_BN_CLICKED(IDC_RADIO_SERVER, OnRadioServer)
      ON_BN_CLICKED(ID_OK, OnOk)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// CInputSelf message handlers

void CInputSelf::OnRadioClient()
{
 // record that we are a client
 type = 0;
}

void CInputSelf::OnRadioServer()
{
 // record that we are a server
 type = 1;
}

void CInputSelf::OnOk()
{
   CDialog::OnOK();
}
```

inputser.h

```
// inputser.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CInputServer dialog

class CInputServer : public CDialog
{
// Construction
public:
       CInputServer(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
       //{{AFX_DATA(CInputServer)
       enum { IDD = IDD_INPUT_SERVER };
       CString  m_input_internet;
       //}}AFX_DATA

// Implementation
protected:
       virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

       // Generated message map functions
       //{{AFX_MSG(CInputServer)
       afx_msg void OnOk();
       //}}AFX_MSG
       DECLARE_MESSAGE_MAP()
};
```

inputser.cpp

```
// inputser.cpp : implementation file
//

#include "stdafx.h"
```

```
#include "netwar2.h"
#include "inputser.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CInputServer dialog


CInputServer::CInputServer(CWnd* pParent /*=NULL*/)
	: CDialog(CInputServer::IDD, pParent)
{
	//{{AFX_DATA_INIT(CInputServer)
	m_input_internet = "";
	//}}AFX_DATA_INIT
}

void CInputServer::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CInputServer)
	DDX_Text(pDX, IDC_INPUT_PLAYER_INTERNET, m_input_internet);
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CInputServer, CDialog)
	//{{AFX_MSG_MAP(CInputServer)
	ON_BN_CLICKED(ID_OK, OnOk)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// CInputServer message handlers

void CInputServer::OnOk()
{
	CDialog::OnOK();
}
```

mainfrm.h

```
// mainfrm.h : interface of the CMainFrame class
//
```

```
/////////////////////////////////////////////////////////////////////////////

class CMainFrame : public CFrameWnd
{

private:
   virtual void ActivateFrame(int nCmdShow = -1);
   BOOL m_bFirstTime;

protected: // create from serialization only
      CMainFrame();
      DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Implementation
public:
      virtual ~CMainFrame();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
      //{{AFX_MSG(CMainFrame)
            // NOTE - the ClassWizard will add and remove member functions here.
            // DO NOT EDIT what you see in these blocks of generated code!
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

mainfrm.cpp

```
// mainfrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "netwar2.h"
#include "mainfrm.h"
```

```
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
      //{{AFX_MSG_MAP(CMainFrame)
            // NOTE - the ClassWizard will add and remove mapping macros here.
            //    DO NOT EDIT what you see in these blocks of generated code !
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
      // TODO: add member initialization code here
      m_bFirstTime = TRUE;
}

CMainFrame::~CMainFrame()
{
}

void CMainFrame::ActivateFrame(int nCmdShow)
{
 CRect rect;
 WINDOWPLACEMENT wndpl;

 if (m_bFirstTime)
 {
  rect.left = 0;
  rect.top = 0;
  rect.right = 500;
  rect.bottom = 480;

  wndpl.length = sizeof(WINDOWPLACEMENT);
  wndpl.showCmd = SW_NORMAL;
  wndpl.flags = 0;
  wndpl.ptMinPosition = CPoint(0,0);
  wndpl.ptMaxPosition = CPoint(-::GetSystemMetrics(SM_CXBORDER),
                               -::GetSystemMetrics(SM_CYBORDER));
  wndpl.rcNormalPosition = rect;
```

```
    BOOL bRect = SetWindowPlacement(&wndpl);
  }
  CFrameWnd::ActivateFrame(nCmdShow);
}


/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
	CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
	CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
```

messagin.h

```
// messagin.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CMessaging dialog

class CMessaging : public CDialog
{
// Construction
public:
	CMessaging(CWnd* pParent = NULL); // standard constructor
	int sendto;

// Dialog Data
	//{{AFX_DATA(CMessaging)
	enum { IDD = IDD_MESSAGE_INPUT };
	CString  m_message_input;
	//}}AFX_DATA

// Implementation
protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

	// Generated message map functions
	//{{AFX_MSG(CMessaging)
	afx_msg void OnSendMessage1();
	afx_msg void OnSendMessage2();
	afx_msg void OnSendMessage3();
	afx_msg void OnSendMessage4();
	afx_msg void OnSendMessageAll();
	virtual void OnCancel();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

messagin.cpp

```
// messagin.cpp : implementation file
//

#include "stdafx.h"
#include "netwar2.h"
#include "messagin.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMessaging dialog


CMessaging::CMessaging(CWnd* pParent /*=NULL*/)
	: CDialog(CMessaging::IDD, pParent)
{
	//{{AFX_DATA_INIT(CMessaging)
	m_message_input = "";
	//}}AFX_DATA_INIT
}

void CMessaging::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CMessaging)
	DDX_Text(pDX, IDC_MESSAGE_INPUT, m_message_input);
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CMessaging, CDialog)
	//{{AFX_MSG_MAP(CMessaging)
	ON_BN_CLICKED(IDC_SEND_MESSAGE_1, OnSendMessage1)
	ON_BN_CLICKED(IDC_SEND_MESSAGE_2, OnSendMessage2)
	ON_BN_CLICKED(IDC_SEND_MESSAGE_3, OnSendMessage3)
	ON_BN_CLICKED(IDC_SEND_MESSAGE_4, OnSendMessage4)
	ON_BN_CLICKED(IDC_SEND_MESSAGE_ALL, OnSendMessageAll)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMessaging message handlers

void CMessaging::OnSendMessage1()
{
 sendto = 1;
 CDialog::OnOK();
}

void CMessaging::OnSendMessage2()
{
 sendto = 2;
 CDialog::OnOK();
}

void CMessaging::OnSendMessage3()
{
 sendto = 3;
 CDialog::OnOK();
}

void CMessaging::OnSendMessage4()
{
 sendto = 4;
 CDialog::OnOK();
}

void CMessaging::OnSendMessageAll()
{
 sendto = 5;
 CDialog::OnOK();
}

void CMessaging::OnCancel()
{
 sendto = 0;
 CDialog::OnCancel();
}
```

netwadoc.h

```
// netwadoc.h : interface of the CNetwar2Doc class
//
/////////////////////////////////////////////////////////////////////////////

class CNetwar2Doc : public CDocument
{
protected: // create from serialization only
	CNetwar2Doc();
	DECLARE_DYNCREATE(CNetwar2Doc)

// Attributes
public:
// Operations
public:

// Implementation
public:
	virtual ~CNetwar2Doc();
	virtual void Serialize(CArchive& ar); // overridden for document i/o
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:
	virtual BOOL OnNewDocument();

// Generated message map functions
protected:
	//{{AFX_MSG(CNetwar2Doc)
	   // NOTE - the ClassWizard will add and remove member functions here.
	   // DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

netwadoc.cpp

```
// netwadoc.cpp : implementation of the CNetwar2Doc class
//

#include "stdafx.h"
```

```
#include "netwar2.h"

#include "netwadoc.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CNetwar2Doc

IMPLEMENT_DYNCREATE(CNetwar2Doc, CDocument)

BEGIN_MESSAGE_MAP(CNetwar2Doc, CDocument)
	//{{AFX_MSG_MAP(CNetwar2Doc)
		// NOTE - the ClassWizard will add and remove mapping macros here.
		// DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CNetwar2Doc construction/destruction

CNetwar2Doc::CNetwar2Doc()
{
	// TODO: add one-time construction code here
}

CNetwar2Doc::~CNetwar2Doc()
{
}

BOOL CNetwar2Doc::OnNewDocument()
{
	if (!CDocument::OnNewDocument())
		return FALSE;

	// TODO: add reinitialization code here
	// (SDI documents will reuse this document)

	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CNetwar2Doc serialization

void CNetwar2Doc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}

/////////////////////////////////////////////////////////////////////////////
// CNetwar2Doc diagnostics

#ifdef _DEBUG
void CNetwar2Doc::AssertValid() const
{
	CDocument::AssertValid();
}

void CNetwar2Doc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CNetwar2Doc commands
```

netwar2.h

```
// netwar2.h : main header file for the NETWAR2 application
//

#define WM_ARMY_SELECTED      WM_USER + 10
#define WM_WAR_BUTTON         WM_USER + 11
#define WM_QUIT_BUTTON        WM_USER + 12
#define MSG_ACCEPT            WM_USER + 13
#define MSG_FROM_PLAYER1      WM_USER + 14
#define MSG_FROM_PLAYER2      WM_USER + 15
#define MSG_FROM_PLAYER3      WM_USER + 16
#define SERVER_MSG            WM_USER + 17

#ifndef __AFXWIN_H__
	#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols
```

```
/////////////////////////////////////////////////////////////////////////////
// CNetwar2App:
// See netwar2.cpp for the implementation of this class
//

class CNetwar2App : public CWinApp
{
public:
	CNetwar2App();

// Overrides
	virtual BOOL InitInstance();

// Implementation

	//{{AFX_MSG(CNetwar2App)
	afx_msg void OnAppAbout();
		// NOTE - the ClassWizard will add and remove member functions here.
		//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////
```

netwavw.h

```
// netwavw.h : interface of the CNetwar2View class
//
/////////////////////////////////////////////////////////////////////////////

#include "sprite.h"
#include "player.h"
#include "warmatrx.h"
#include "server.h"
#include "winsock.h"
#include "serverwa.h"

#define WM_ARMY_SELECTED WM_USER + 10


#define CAVALRY_BITMAP              0
#define CAVALRY_BITMAP_MASK         1
#define CAVALRY_BITMAP_HIGHLIGHT    2

#define FLYER_BITMAP                3
```

```
#define FLYER_BITMAP_MASK             4
#define FLYER_BITMAP_HIGHLIGHT        5

#define ARCHER_BITMAP                 6
#define ARCHER_BITMAP_MASK            7
#define ARCHER_BITMAP_HIGHLIGHT       8

#define LEGION_BITMAP                 9
#define LEGION_BITMAP_MASK           10
#define LEGION_BITMAP_HIGHLIGHT      11

#define NETWORK_NONE                  0
#define NETWORK_INTERNET              1
#define NETWORK_SERIAL                2
#define NETWORK_WORKGROUPS            3


class operations;

class CNetwar2View : public CView
{
private:
   CDC*      m_pDisplayMemDC;
   CBitmap*  playing_field_bitmap;

   CBitmap*  cavalry_bitmap;
   CBitmap*  cavalry_bitmap_mask;
   CBitmap*  cavalry_bitmap_highlight;
   CBitmap*  bitmaps[4][12];


   CSprite*  cavalry_piece;
   int       dragging;

   int       current_select;

   operations* ops_dlg;

//players
   CPlayer*  players[4];
   int       who_am_i;
   int       total_players;

//sprites
   CSprite*  playing_pieces[4][50];
   int   total_pieces[4];

//war engine
   CBattle*  war_matrix;
   int       war_button;
```

```
    int        war_timer;
    int        lost_token_timer;
    int        first_clicked;
    int        second_clicked_player,
               second_clicked_icon;

//Sound
     HRSRC add_army_sound;
     HGLOBAL add_army_sound_load;

     BOOL do_sound;

//Server
   CServer*  the_server;
   int       server;

//Networks
     int          current_network;
     WSADATA      wsaData;
     char         send_buffer[70];
     CServerWait msg_dlg;

   int gold_mine_timer;
   CRect* gold_mine_rect;

   BOOL DO_EXTRA_INVAL;

   int COMPortID;
   char serial_initialization_string[25];
   int serial_com_port;
   int serial_baud_rate;

    int send_timer;
    int total_acks;
    BOOL time_up;
    BOOL ACTIVATE_ACKS;

protected: // create from serialization only
    CNetwar2View();
    DECLARE_DYNCREATE(CNetwar2View)

// Attributes
public:
    CNetwar2Doc* GetDocument();

// Operations
public:

// Implementation
public:
    virtual ~CNetwar2View();
    virtual void OnDraw(CDC* pDC); // overridden to draw this view
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CNetwar2View)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnRButtonDblClk(UINT nFlags, CPoint point);
    afx_msg void OnPlayersBuyequipment();
    afx_msg void OnOptionsSound();
    afx_msg void OnUpdateOptionsSound(CCmdUI* pCmdUI);
    afx_msg void OnFileNew();
    afx_msg void OnPlayersInputplayers();
    afx_msg void OnNetworkSerialmodem();
    afx_msg void OnUpdateNetworkSerialmodem(CCmdUI* pCmdUI);
    afx_msg void OnUpdateNetworkWindowsforworkgroups(CCmdUI* pCmdUI);
    afx_msg void OnNetworkWindowsforworkgroups();
    afx_msg void OnNetworkWinsockinternet();
    afx_msg void OnUpdateNetworkWinsockinternet(CCmdUI* pCmdUI);
    afx_msg void OnWaitforconnect();
    afx_msg void OnConnecttoserver();
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnOptionsExtrainvalidate();
    afx_msg void OnUpdateOptionsExtrainvalidate(CCmdUI* pCmdUI);
    afx_msg void OnOptionsAcknowledgements();
    afx_msg void OnUpdateOptionsAcknowledgements(CCmdUI* pCmdUI);
    //}}AFX_MSG
    long OnArmySelected(UINT wParam, LONG lParam);
    long OnWar(UINT wParam, LONG lParam);
    long OnQuitButton(UINT wParam, LONG lParam);
    long OnMsgAccept(UINT wParam, LONG lParam);
    void SendInfo(char *command, int data1, int data2, int data3, char *msg);
    long OnMsgFromPlayer1(UINT wParam, LONG lParam);
    long OnMsgFromPlayer2(UINT wParam, LONG lParam);
    long OnMsgFromPlayer3(UINT wParam, LONG lParam);
    long SendToOthers(int client, char *buffer);
    long DoCommand(char *buffer);
    long OnServerSend(UINT wParam, LONG lParam);
    long OnMessageButton(UINT wParam, LONG lParam);
    long OnCommNotify(UINT wParam, LONG lParam);
    void send_ack(int from);
    void get_ack();
    DECLARE_MESSAGE_MAP()
```

```
};

#ifndef _DEBUG // debug version in netwavw.cpp
inline CNetwar2Doc* CNetwar2View::GetDocument()
   { return (CNetwar2Doc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
```

netwavw.cpp

```
// netwavw.cpp : implementation of the CNetwar2View class
//

#include "stdafx.h"
#include "netwar2.h"
#include "math.h"

#include "netwadoc.h"
#include "netwavw.h"

#include "ppstats.h"
#include "operatio.h"
#include "buy_equi.h"
#include "inputsel.h"
#include "inputser.h"
#include "messagin.h"
#include "omminput.h"

#include <mmsystem.h>

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CNetwar2View

IMPLEMENT_DYNCREATE(CNetwar2View, CView)

BEGIN_MESSAGE_MAP(CNetwar2View, CView)
      ON_MESSAGE(WM_ARMY_SELECTED, OnArmySelected)
      ON_MESSAGE(WM_QUIT_BUTTON, OnQuitButton)
      ON_MESSAGE(MSG_ACCEPT, OnMsgAccept)
      ON_MESSAGE(MSG_FROM_PLAYER1, OnMsgFromPlayer1)
      ON_MESSAGE(MSG_FROM_PLAYER2, OnMsgFromPlayer2)
      ON_MESSAGE(MSG_FROM_PLAYER3, OnMsgFromPlayer3)
```

```
	ON_MESSAGE(SERVER_MSG, OnServerSend)
	ON_MESSAGE(WM_WAR_BUTTON, OnWar)
	ON_MESSAGE(WM_MESSAGE_BUTTON, OnMessageButton)
	//{{AFX_MSG_MAP(CNetwar2View)
	ON_WM_LBUTTONDOWN()
	ON_WM_MOUSEMOVE()
	ON_WM_LBUTTONDBLCLK()
	ON_WM_LBUTTONUP()
	ON_WM_RBUTTONDBLCLK()
	ON_COMMAND(ID_PLAYERS_BUYEQUIPMENT, OnPlayersBuyequipment)
	ON_COMMAND(ID_OPTIONS_SOUND, OnOptionsSound)
	ON_UPDATE_COMMAND_UI(ID_OPTIONS_SOUND, OnUpdateOptionsSound)
	ON_COMMAND(ID_FILE_NEW, OnFileNew)
	ON_COMMAND(ID_PLAYERS_INPUTPLAYERS, OnPlayersInputplayers)
	ON_COMMAND(ID_NETWORK_SERIALMODEM, OnNetworkSerialmodem)
	ON_UPDATE_COMMAND_UI(ID_NETWORK_SERIALMODEM, OnUpdateNetworkSerialmodem)
	ON_UPDATE_COMMAND_UI(ID_NETWORK_WINDOWSFORWORKGROUPS, OnUpdateNetworkWindows-
forworkgroups)
	ON_COMMAND(ID_NETWORK_WINDOWSFORWORKGROUPS, OnNetworkWindowsforworkgroups)
	ON_COMMAND(ID_NETWORK_WINSOCKINTERNET, OnNetworkWinsockinternet)
	ON_UPDATE_COMMAND_UI(ID_NETWORK_WINSOCKINTERNET, OnUpdateNetworkWinsockinternet)
	ON_COMMAND(ID_WAITFORCONNECT, OnWaitforconnect)
	ON_COMMAND(ID_CONNECTTOSERVER, OnConnecttoserver)
	ON_WM_TIMER()
	ON_COMMAND(ID_OPTIONS_EXTRAINVALIDATE, OnOptionsExtrainvalidate)
	ON_UPDATE_COMMAND_UI(ID_OPTIONS_EXTRAINVALIDATE, OnUpdateOptionsExtrainvalidate)
	ON_COMMAND(ID_OPTIONS_ACKNOWLEDGEMENTS, OnOptionsAcknowledgements)
	ON_UPDATE_COMMAND_UI(ID_OPTIONS_ACKNOWLEDGEMENTSOnUpdateOptionsAcknowledgements)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CNetwar2View construction/destruction

CNetwar2View::CNetwar2View()
{
 int i;

 m_pDisplayMemDC = new CDC;
 CClientDC dc(this);
 OnPrepareDC(&dc);
 m_pDisplayMemDC->CreateCompatibleDC(&dc);

 // Load Playing field bitmap
 playing_field_bitmap = new CBitmap;
 if (playing_field_bitmap->LoadBitmap(IDB_PLAYING_FIELD) == FALSE)
  MessageBox("Unable to load bitmap", "error", MB_OK);

 m_pDisplayMemDC->SelectObject(playing_field_bitmap);

 //Load bitmaps
```

```
bitmaps[0][CAVALRY_BITMAP] = new CBitmap;
if (bitmaps[0][CAVALRY_BITMAP]->LoadBitmap(IDB_P0_CAVALRY_BITMAP) == FALSE)
 MessageBox(“Unable to load cavalry bitmap”, “error”, MB_OK);

bitmaps[0][CAVALRY_BITMAP_MASK] = new CBitmap;
if (bitmaps[0][CAVALRY_BITMAP_MASK]->LoadBitmap(IDB_P0_CAVALRY_MASK) == FALSE)
 MessageBox(“Unable to load cavalry mask bitmap”, “error”, MB_OK);

bitmaps[0][CAVALRY_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[0][CAVALRY_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P0_CAVALRY_HL) == FALSE)
 MessageBox(“Unable to load cavalry highlight bitmap”, “error”, MB_OK);

bitmaps[0][FLYER_BITMAP] = new CBitmap;
if (bitmaps[0][FLYER_BITMAP]->LoadBitmap(IDB_P0_FLYER_BITMAP) == FALSE)
 MessageBox(“Unable to load flyer bitmap”, “error”, MB_OK);

bitmaps[0][FLYER_BITMAP_MASK] = new CBitmap;
if (bitmaps[0][FLYER_BITMAP_MASK]->LoadBitmap(IDB_P0_FLYER_MASK) == FALSE)
 MessageBox(“Unable to load flyer mask bitmap”, “error”, MB_OK);

bitmaps[0][FLYER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[0][FLYER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P0_FLYER_HL) == FALSE)
 MessageBox(“Unable to load flyer highlight bitmap”, “error”, MB_OK);

bitmaps[0][ARCHER_BITMAP] = new CBitmap;
if (bitmaps[0][ARCHER_BITMAP]->LoadBitmap(IDB_P0_ARCHER_BITMAP) == FALSE)
 MessageBox(“Unable to load archer bitmap”, “error”, MB_OK);

bitmaps[0][ARCHER_BITMAP_MASK] = new CBitmap;
if (bitmaps[0][ARCHER_BITMAP_MASK]->LoadBitmap(IDB_P0_ARCHER_MASK) == FALSE)
 MessageBox(“Unable to load archer mask bitmap”, “error”, MB_OK);

bitmaps[0][ARCHER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[0][ARCHER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P0_ARCHER_HL) == FALSE)
 MessageBox(“Unable to load archer highlight bitmap”, “error”, MB_OK);

bitmaps[0][LEGION_BITMAP] = new CBitmap;
if (bitmaps[0][LEGION_BITMAP]->LoadBitmap(IDB_P0_LEGION_BITMAP) == FALSE)
 MessageBox(“Unable to load legion bitmap”, “error”, MB_OK);

bitmaps[0][LEGION_BITMAP_MASK] = new CBitmap;
if (bitmaps[0][LEGION_BITMAP_MASK]->LoadBitmap(IDB_P0_LEGION_MASK) == FALSE)
 MessageBox(“Unable to load legion mask bitmap”, “error”, MB_OK);

bitmaps[0][LEGION_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[0][LEGION_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P0_LEGION_HL) == FALSE)
 MessageBox(“Unable to load legion highlight bitmap”, “error”, MB_OK);


bitmaps[1][CAVALRY_BITMAP] = new CBitmap;
if (bitmaps[1][CAVALRY_BITMAP]->LoadBitmap(IDB_P1_CAVALRY_BITMAP) == FALSE)
```

```
  MessageBox("Unable to load cavalry bitmap", "error", MB_OK);

bitmaps[1][CAVALRY_BITMAP_MASK] = new CBitmap;
if (bitmaps[1][CAVALRY_BITMAP_MASK]->LoadBitmap(IDB_P1_CAVALRY_MASK) == FALSE)
  MessageBox("Unable to load cavalry mask bitmap", "error", MB_OK);

bitmaps[1][CAVALRY_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[1][CAVALRY_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P1_CAVALRY_HL) == FALSE)
  MessageBox("Unable to load cavalry highlight bitmap", "error", MB_OK);

bitmaps[1][FLYER_BITMAP] = new CBitmap;
if (bitmaps[1][FLYER_BITMAP]->LoadBitmap(IDB_P1_FLYER_BITMAP) == FALSE)
  MessageBox("Unable to load flyer bitmap", "error", MB_OK);

bitmaps[1][FLYER_BITMAP_MASK] = new CBitmap;
if (bitmaps[1][FLYER_BITMAP_MASK]->LoadBitmap(IDB_P1_FLYER_MASK) == FALSE)
  MessageBox("Unable to load flyer mask bitmap", "error", MB_OK);

bitmaps[1][FLYER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[1][FLYER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P1_FLYER_HL) == FALSE)
  MessageBox("Unable to load flyer highlight bitmap", "error", MB_OK);

bitmaps[1][ARCHER_BITMAP] = new CBitmap;
if (bitmaps[1][ARCHER_BITMAP]->LoadBitmap(IDB_P1_ARCHER_BITMAP) == FALSE)
  MessageBox("Unable to load archer bitmap", "error", MB_OK);

bitmaps[1][ARCHER_BITMAP_MASK] = new CBitmap;
if (bitmaps[1][ARCHER_BITMAP_MASK]->LoadBitmap(IDB_P1_ARCHER_MASK) == FALSE)
  MessageBox("Unable to load archer mask bitmap", "error", MB_OK);

bitmaps[1][ARCHER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[1][ARCHER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P1_ARCHER_HL) == FALSE)
  MessageBox("Unable to load archer highlight bitmap", "error", MB_OK);

bitmaps[1][LEGION_BITMAP] = new CBitmap;
if (bitmaps[1][LEGION_BITMAP]->LoadBitmap(IDB_P1_LEGION_BITMAP) == FALSE)
  MessageBox("Unable to load legion bitmap", "error", MB_OK);

bitmaps[1][LEGION_BITMAP_MASK] = new CBitmap;
if (bitmaps[1][LEGION_BITMAP_MASK]->LoadBitmap(IDB_P1_LEGION_MASK) == FALSE)
  MessageBox("Unable to load legion mask bitmap", "error", MB_OK);

bitmaps[1][LEGION_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[1][LEGION_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P1_LEGION_HL) == FALSE)
  MessageBox("Unable to load legion highlight bitmap", "error", MB_OK);

bitmaps[2][CAVALRY_BITMAP] = new CBitmap;
if (bitmaps[2][CAVALRY_BITMAP]->LoadBitmap(IDB_P2_CAVALRY_BITMAP) == FALSE)
  MessageBox("Unable to load cavalry bitmap", "error", MB_OK);

bitmaps[2][CAVALRY_BITMAP_MASK] = new CBitmap;
```

```
if (bitmaps[2][CAVALRY_BITMAP_MASK]->LoadBitmap(IDB_P2_CAVALRY_MASK) == FALSE)
 MessageBox("Unable to load cavalry mask bitmap", "error", MB_OK);

bitmaps[2][CAVALRY_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[2][CAVALRY_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P2_CAVALRY_HL) == FALSE)
 MessageBox("Unable to load cavalry highlight bitmap", "error", MB_OK);

bitmaps[2][FLYER_BITMAP] = new CBitmap;
if (bitmaps[2][FLYER_BITMAP]->LoadBitmap(IDB_P2_FLYER_BITMAP) == FALSE)
 MessageBox("Unable to load flyer bitmap", "error", MB_OK);

bitmaps[2][FLYER_BITMAP_MASK] = new CBitmap;
if (bitmaps[2][FLYER_BITMAP_MASK]->LoadBitmap(IDB_P2_FLYER_MASK) == FALSE)
 MessageBox("Unable to load flyer mask bitmap", "error", MB_OK);

bitmaps[2][FLYER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[2][FLYER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P2_FLYER_HL) == FALSE)
 MessageBox("Unable to load flyer highlight bitmap", "error", MB_OK);

bitmaps[2][ARCHER_BITMAP] = new CBitmap;
if (bitmaps[2][ARCHER_BITMAP]->LoadBitmap(IDB_P2_ARCHER_BITMAP) == FALSE)
 MessageBox("Unable to load archer bitmap", "error", MB_OK);

bitmaps[2][ARCHER_BITMAP_MASK] = new CBitmap;
if (bitmaps[2][ARCHER_BITMAP_MASK]->LoadBitmap(IDB_P2_ARCHER_MASK) == FALSE)
 MessageBox("Unable to load archer mask bitmap", "error", MB_OK);

bitmaps[2][ARCHER_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[2][ARCHER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P2_ARCHER_HL) == FALSE)
 MessageBox("Unable to load archer highlight bitmap", "error", MB_OK);

bitmaps[2][LEGION_BITMAP] = new CBitmap;
if (bitmaps[2][LEGION_BITMAP]->LoadBitmap(IDB_P2_LEGION_BITMAP) == FALSE)
 MessageBox("Unable to load legion bitmap", "error", MB_OK);

bitmaps[2][LEGION_BITMAP_MASK] = new CBitmap;
if (bitmaps[2][LEGION_BITMAP_MASK]->LoadBitmap(IDB_P2_LEGION_MASK) == FALSE)
 MessageBox("Unable to load legion mask bitmap", "error", MB_OK);

bitmaps[2][LEGION_BITMAP_HIGHLIGHT] = new CBitmap;
if (bitmaps[2][LEGION_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P2_LEGION_HL) == FALSE)
 MessageBox("Unable to load legion highlight bitmap", "error", MB_OK);


bitmaps[3][CAVALRY_BITMAP] = new CBitmap;
if (bitmaps[3][CAVALRY_BITMAP]->LoadBitmap(IDB_P3_CAVALRY_BITMAP) == FALSE)
 MessageBox("Unable to load cavalry bitmap", "error", MB_OK);

bitmaps[3][CAVALRY_BITMAP_MASK] = new CBitmap;
if (bitmaps[3][CAVALRY_BITMAP_MASK]->LoadBitmap(IDB_P3_CAVALRY_MASK) == FALSE)
 MessageBox("Unable to load cavalry mask bitmap", "error", MB_OK);
```

```
    bitmaps[3][CAVALRY_BITMAP_HIGHLIGHT] = new CBitmap;
    if (bitmaps[3][CAVALRY_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P3_CAVALRY_HL) == FALSE)
     MessageBox("Unable to load cavalry highlight bitmap", "error", MB_OK);

    bitmaps[3][FLYER_BITMAP] = new CBitmap;
    if (bitmaps[3][FLYER_BITMAP]->LoadBitmap(IDB_P3_FLYER_BITMAP) == FALSE)
     MessageBox("Unable to load flyer bitmap", "error", MB_OK);

    bitmaps[3][FLYER_BITMAP_MASK] = new CBitmap;
    if (bitmaps[3][FLYER_BITMAP_MASK]->LoadBitmap(IDB_P3_FLYER_MASK) == FALSE)
     MessageBox("Unable to load flyer mask bitmap", "error", MB_OK);

    bitmaps[3][FLYER_BITMAP_HIGHLIGHT] = new CBitmap;
    if (bitmaps[3][FLYER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P3_FLYER_HL) == FALSE)
     MessageBox("Unable to load flyer highlight bitmap", "error", MB_OK);

    bitmaps[3][ARCHER_BITMAP] = new CBitmap;
    if (bitmaps[3][ARCHER_BITMAP]->LoadBitmap(IDB_P3_ARCHER_BITMAP) == FALSE)
     MessageBox("Unable to load archer bitmap", "error", MB_OK);

    bitmaps[3][ARCHER_BITMAP_MASK] = new CBitmap;
    if (bitmaps[3][ARCHER_BITMAP_MASK]->LoadBitmap(IDB_P3_ARCHER_MASK) == FALSE)
     MessageBox("Unable to load archer mask bitmap", "error", MB_OK);

    bitmaps[3][ARCHER_BITMAP_HIGHLIGHT] = new CBitmap;
    if (bitmaps[3][ARCHER_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P3_ARCHER_HL) == FALSE)
     MessageBox("Unable to load archer highlight bitmap", "error", MB_OK);

    bitmaps[3][LEGION_BITMAP] = new CBitmap;
    if (bitmaps[3][LEGION_BITMAP]->LoadBitmap(IDB_P3_LEGION_BITMAP) == FALSE)
     MessageBox("Unable to load legion bitmap", "error", MB_OK);

    bitmaps[3][LEGION_BITMAP_MASK] = new CBitmap;
    if (bitmaps[3][LEGION_BITMAP_MASK]->LoadBitmap(IDB_P3_LEGION_MASK) == FALSE)
     MessageBox("Unable to load legion mask bitmap", "error", MB_OK);

    bitmaps[3][LEGION_BITMAP_HIGHLIGHT] = new CBitmap;
    if (bitmaps[3][LEGION_BITMAP_HIGHLIGHT]->LoadBitmap(IDB_P3_LEGION_HL) == FALSE)
     MessageBox("Unable to load legion highlight bitmap", "error", MB_OK);


  //cavalry_piece = new CSprite;
  //if (cavalry_piece->Initialize(GetDC(), cavalry_bitmap_mask, cavalry_bitmap,
cavalry_bitmap_highlight, FALSE, 10, 10, 1, "Cavalry", 5) == FALSE)
// MessageBox("Unable to load bitmap", "error", MB_OK);

  dragging = -1;
  current_select = 0;

  //create operations dialog box
  ops_dlg = new operations(this);
  if (ops_dlg->GetSafeHwnd() == 0)
    ops_dlg->Create();

  total_players = 0;
  for (i=0;i<4;i++)
    total_pieces[i] = 0;

  // Create war matrix
  war_matrix = new CBattle;

  //Load sounds
  add_army_sound = ::FindResource(AfxGetResourceHandle(),
(LPCSTR)MAKEINTRESOURCE(ADD_ARMY_SOUND), "sound");
  add_army_sound_load = ::LoadResource(AfxGetResourceHandle(), add_army_sound);

  do_sound = FALSE;

  //network setup
  current_network = NETWORK_NONE;

  who_am_i = -1;

  DO_EXTRA_INVAL = FALSE;
  ACTIVATE_ACKS = FALSE;
}


CNetwar2View::~CNetwar2View()
{
  int i;

  if (current_network == NETWORK_INTERNET)
  {
    for (i=0;i<total_players;i++)
    {
      if (server == 0)
      {
        if (the_server->active_connections[i] == TRUE)
          closesocket(the_server->clients[i]);
    }
    else
      closesocket(players[0]->socket);
  }
  while (WSACleanup() != 0);
  }
  //for (i=0;i<3;i++)
  // delete playing_pieces[i];

  //delete cavalry_bitmap;
  //delete cavalry_bitmap_mask;
  //delete cavalry_bitmap_highlight;
```

```
   //delete cavalry piece;
   delete playing_field_bitmap;
  }


/////////////////////////////////////////////////////////////////////////////
// CNetwar2View drawing

void CNetwar2View::OnDraw(CDC* pDC)
{
 BITMAP bm;
 int i, j;

 playing_field_bitmap->GetObject(sizeof(bm), &bm);
 pDC->BitBlt(0,0,bm.bmWidth, bm.bmHeight, m_pDisplayMemDC, 0, 0, SRCCOPY);

 for (j=0;j<total_players;j++)
 {
  for(i=0;i<total_pieces[j];i++)
  {
   if (playing_pieces[j][i]->active == TRUE)
   {
    if (playing_pieces[j][i]->show)
    {
     playing_pieces[j][i]->Redraw(pDC);
     }
     else
     {
      switch (playing_pieces[j][i]->player)
      {
       case 0: playing_pieces[j][i]->Start(pDC, 50, 50);
               break;

      case 1: playing_pieces[j][i]->Start(pDC, 400, 50);
              break;

      case 2: playing_pieces[j][i]->Start(pDC, 50, 350);
              break;

      case 3: playing_pieces[j][i]->Start(pDC, 400, 350);
              break;
     }
     playing_pieces[j][i]->show = TRUE;
    }
   }
  }
 }

}
/////////////////////////////////////////////////////////////////////////////
// CNetwar2View diagnostics
```

```
#ifdef _DEBUG
void CNetwar2View::AssertValid() const
{
      CView::AssertValid();
}

void CNetwar2View::Dump(CDumpContext& dc) const
{
      CView::Dump(dc);
}

CNetwar2Doc* CNetwar2View::GetDocument() // non-debug version is inline
{
      ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CNetwar2Doc)));
      return (CNetwar2Doc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CNetwar2View message handlers

void CNetwar2View::OnLButtonDown(UINT nFlags, CPoint point)
{
 CRect* SpriteRect;
 CRect clientRect;
 CPoint topleft;
 int i;

 for (i=0;i<total_pieces[who_am_i];i++)
 {
  if (playing_pieces[who_am_i][i]->active == TRUE)
  {
   SpriteRect = new CRect(playing_pieces[who_am_i][i]->mX, playing_pieces[who_am_i]
[i]->mY,
   playing_pieces[who_am_i][i]->mX+playing_pieces[who_am_i][i]->mWidth,
playing_pieces[who_am_i][i]->mY+playing_pieces[who_am_i][i]->mHeight);

   if ((playing_pieces[who_am_i][i]->war == FALSE) && (SpriteRect-
>PtInRect(point)))
   {
    SetCapture();

    GetClientRect(clientRect);
    MapWindowPoints(NULL, clientRect);
    topleft = SpriteRect->TopLeft();

    playing_pieces[who_am_i][i]->XOffset = point.x - topleft.x;
    playing_pieces[who_am_i][i]->YOffset = point.y - topleft.y;
    playing_pieces[who_am_i][i]->startx = point.x;
    playing_pieces[who_am_i][i]->starty = point.y;
```

```
        dragging = i;
        return;
      }
    }
  }
}

void CNetwar2View::OnMouseMove(UINT nFlags, CPoint point)
{
  int move_x, move_y,length, cost;

  if (dragging == -1)
    return;

  move_x = (point.x - playing_pieces[who_am_i][dragging]->XOffset);
  move_y = (point.y - playing_pieces[who_am_i][dragging]->YOffset);

  length = (int)sqrt(pow((double)point.x-playing_pieces[who_am_i][dragging]->startx,
2) + pow((double)point.y-playing_pieces[who_am_i][dragging]->starty, 2));
  cost = length * playing_pieces[who_am_i][dragging]->speed; // *
(1+(playing_pieces[who_am_i][dragging]->health/100));

  if ((length < playing_pieces[who_am_i][dragging]->speed) && (cost < players[0]-
>gold_coins))
  {
    players[0]->gold_coins -= cost;
    ops_dlg->total_gold = players[0]->gold_coins;
    ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);

    playing_pieces[who_am_i][dragging]->MoveTo(GetDC(), move_x, move_y);

    if (DO_EXTRA_INVAL) Invalidate(TRUE);
  }
}

void CNetwar2View::OnLButtonDblClk(UINT nFlags, CPoint point)
{

  CRect* SpriteRect;
  int i, j;

  if (war_button == who_am_i)
  {
  for(i=0;i<total_players;i++)
  {
    for(j=0;j<total_pieces[i];j++)
    {

        SpriteRect = new CRect(playing_pieces[i][j]->mX, playing_pieces[i][j]->mY,
                         playing_pieces[i][j]->mX+playing_pieces[i][j]->mWidth,
playing_pieces[i][j]->mY+playing_pieces[i][j]->mHeight);

      if (SpriteRect->PtInRect(point))
      {

        if ((current_select == 1) && (i == who_am_i) && (first_clicked == j))
        {
         current_select = 0;
         SendInfo("sprite_unhi", i, j, 0, "");

         playing_pieces[i][j]->ReplaceBitmap();
         playing_pieces[i][j]->war = FALSE;

         Invalidate(TRUE);
         return;
        }


        if ((current_select == 0) && (i == who_am_i) && (playing_pieces[i][j]->war ==
FALSE))
        {
         first_clicked = j;
         current_select = 1;

         SendInfo("sprite_high", i, j, 0, "");

         playing_pieces[i][j]->ReplaceBitmap();
         playing_pieces[i][j]->war = TRUE;
         Invalidate(TRUE);
         return;
        }


        if ((current_select == 1) && (i != who_am_i) && (playing_pieces[i][j]->war ==
FALSE))
        {
         SendInfo("sprite_high", i, j, 0, "");

         second_clicked_player = i;
         second_clicked_icon = j;

         playing_pieces[i][j]->war = TRUE;

         current_select = 2;

         playing_pieces[i][j]->ReplaceBitmap();
         Invalidate(TRUE);
         return;
        }
      }
    }
    }
  }
}


void CNetwar2View::OnLButtonUp(UINT nFlags, CPoint point)
{
  if (dragging == -1)
  return;

  ReleaseCapture();

  SendInfo("sprite_stop", playing_pieces[who_am_i][dragging]->mX,
playing_pieces[who_am_i][dragging]->mY, dragging, "");

  dragging = -1;
}

void CNetwar2View::OnRButtonDblClk(UINT nFlags, CPoint point)
{
  PPSTATS     Dlg;
  CRect*      SpriteRect;
  char        string[25];
  int i;

  for (i=0;i<total_pieces[who_am_i];i++)
  {
   if (playing_pieces[who_am_i][i]->active == TRUE)
   {
    SpriteRect = new CRect(playing_pieces[who_am_i][i]->mX,
playing_pieces[who_am_i][i]->mY,
    playing_pieces[who_am_i][i]->mX+playing_pieces[who_am_i][i]->mWidth,
playing_pieces[who_am_i][i]->mY+playing_pieces[who_am_i][i]->mHeight);


    if (SpriteRect->PtInRect(point))
    {
     // fill in player
     sprintf(string, "Owner: Player %d", playing_pieces[who_am_i][i]->player);
     Dlg.m_stats_owner = string;

     // fill in type
     sprintf(string, "Type: %s", playing_pieces[who_am_i][i]->type);
     Dlg.m_stats_type = string;

     // fill in health
     sprintf(string, "Defense: %d units", playing_pieces[who_am_i][i]->defense);
     Dlg.m_stats_defense = string;

     // fill in strength
     sprintf(string, "Offense: %d units", playing_pieces[who_am_i][i]->offense);
```

```
      Dlg.m_stats_offense = string;

      int ret = Dlg.DoModal();
      return;
     }
   }
  }
}


long CNetwar2View::OnArmySelected(UINT wParam, LONG lParam)
{
 SendInfo("sprite_show", ops_dlg->selected_army, 0, 0, "");

 playing_pieces[who_am_i][ops_dlg->selected_army]->active = TRUE;
 Invalidate(TRUE);

 if (do_sound)
 {

sndPlaySound((LPCSTR)::LockResource(add_army_sound_load),SND_MEMORY|SND_ASYNC|SND_NO
DEFAULT);
   UnlockResource(add_army_sound_load);
 }


 return 1;
}


long CNetwar2View::OnQuitButton(UINT wParam, LONG lParam)
{
 int i;

 for(i=0;i<total_pieces[who_am_i]; i++)
   playing_pieces[who_am_i][i]->active = FALSE;

 total_pieces[who_am_i] = 0;

 sprintf(ops_dlg->new_message, "Player %d has surrendered", who_am_i,
second_clicked_player);
 ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);

 ops_dlg->SendMessage(WM_OPERATIONS_PLAYER_QUITS);

 SendInfo("sprite_quit", who_am_i, 0, 0, "");
 SendInfo("print_str", 7, who_am_i, who_am_i, "");

 Invalidate(TRUE);

 return 1;
}
```

```
void CNetwar2View::OnPlayersBuyequipment()
{
 int i;
 buy_equipment pDlg;


 if (who_am_i != -1)
 {
 pDlg.total_gold_coins = players[0]->gold_coins;

 pDlg.DoModal();

 players[0]->gold_coins = pDlg.total_gold_coins;

 ops_dlg->total_gold = pDlg.total_gold_coins;
 ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);

 if (pDlg.cavalrys_bought > 0)
 {
     for(i=0;i<pDlg.cavalrys_bought;i++)
     {
    playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
    if (playing_pieces[who_am_i][total_pieces[who_am_i]]->Initialize(GetDC(),
bitmaps[who_am_i][CAVALRY_BITMAP_MASK], bitmaps[who_am_i][CAVALRY_BITMAP],
                                 bitmaps[who_am_i][CAVALRY_BITMAP_HIGHLIGHT],
                       FALSE, 4, 3, who_am_i, "Cavalry", 40) == FALSE)
      MessageBox("Unable to initialize dragon sprite", "Error", MB_OK);

    strcpy(ops_dlg->entries[ops_dlg->total_armies], "Cavalry");
    ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
    ops_dlg->total_armies++;

    total_pieces[who_am_i]++;
   }
   SendInfo("create sprt", 1, pDlg.cavalrys_bought, 0, "");
 }

  if (pDlg.flyers_bought > 0)
  {
     for(i=0;i<pDlg.flyers_bought;i++)
     {
    playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
    if (playing_pieces[who_am_i][total_pieces[who_am_i]]->Initialize(GetDC(),
bitmaps[who_am_i][FLYER_BITMAP_MASK], bitmaps[who_am_i][FLYER_BITMAP],
                                 bitmaps[who_am_i][FLYER_BITMAP_HIGHLIGHT],
                       FALSE, 3, 3, who_am_i, "Flyer", 90) == FALSE)
      MessageBox("Unable to initialize flyer sprite", "Error", MB_OK);

    strcpy(ops_dlg->entries[ops_dlg->total_armies], "Flyer");
    ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
    ops_dlg->total_armies++;
```

```
      total_pieces[who_am_i]++;
     }
    SendInfo("create_sprt", 4, pDlg.flyers_bought, 0, "");
   }

 if (pDlg.archers_bought > 0)
  {
     for(i=0;i<pDlg.archers_bought;i++)
     {
    playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
    if (playing_pieces[who_am_i][total_pieces[who_am_i]]->Initialize(GetDC(),
 bitmaps[who_am_i][ARCHER_BITMAP_MASK], bitmaps[who_am_i][ARCHER_BITMAP],
                                        bitmaps[who_am_i][ARCHER_BITMAP_HIGHLIGHT],
                         FALSE, 4, 1, who_am_i, "Archer", 30) == FALSE)
      MessageBox("Unable to initialize archer sprite", "Error", MB_OK);

     strcpy(ops_dlg->entries[ops_dlg->total_armies], "Archery");
     ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
     ops_dlg->total_armies++;

     total_pieces[who_am_i]++;
    }
   SendInfo("create_sprt", 2, pDlg.archers_bought, 0, "");
  }

 if (pDlg.legions_bought > 0)
  {
      for(i=0;i<pDlg.legions_bought;i++)
      {
    playing_pieces[who_am_i][total_pieces[who_am_i]] = new CSprite;
    if (playing_pieces[who_am_i][total_pieces[who_am_i]]->Initialize(GetDC(),
 bitmaps[who_am_i][LEGION_BITMAP_MASK], bitmaps[who_am_i][LEGION_BITMAP],
                                        bitmaps[who_am_i][LEGION_BITMAP_HIGHLIGHT],
                         FALSE, 8, 6, who_am_i, "Legion", 20) == FALSE)
      MessageBox("Unable to initialize legion sprite", "Error", MB_OK);

     strcpy(ops_dlg->entries[ops_dlg->total_armies], "Legion");
     ops_dlg->list_box[ops_dlg->total_armies] = total_pieces[who_am_i];
     ops_dlg->total_armies++;

     total_pieces[who_am_i]++;
    }
   SendInfo("create_sprt", 3, pDlg.legions_bought, 0, "");
  }

     ops_dlg->SendMessage(WM_OPERATIONS_ADD_ARMIES, 0, 0L);
    }
   else
     MessageBox("Enter Player Information and Connect to Server First", MB_OK);
}

void CNetwar2View::OnOptionsSound()
```

```
{
      if (do_sound)
       do_sound = FALSE;
      else
       do_sound = TRUE;

}

void CNetwar2View::OnUpdateOptionsSound(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(do_sound);
}

void CNetwar2View::OnFileNew()
{
 int i;

 SendInfo("game_new", who_am_i, 0, 0, "");

 ops_dlg->SendMessage(WM_OPERATIONS_NEW_GAME, 0, 0L);
 for (i=0;i<total_pieces[0];i++)
  delete playing_pieces[0][i];

 total_pieces[0] = 0;

 Invalidate(TRUE);
 players[0]->gold_coins = 500000;
 players[0]->kills = 0;
}

void CNetwar2View::OnPlayersInputplayers()
{
 CInputSelf Self_Dlg;
 CInputServer Server_Dlg;

 if (total_players == 0)
 {
     total_players = 1;
     players[0] = new CPlayer;

     Self_Dlg.DoModal();
     total_players = atoi(Self_Dlg.m_total_players);

     //set gold in operations window
     ops_dlg->total_gold = 500000;
    ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);

    the_server = new CServer;
     if (Self_Dlg.type == 1)
     {
      server = 0;
      who_am_i = 0;
  }
  else
  {
  int ret = Server_Dlg.DoModal();

  strcpy(the_server->address, Server_Dlg.m_input_internet);
     server = 1;
     }
  }
}

void CNetwar2View::OnNetworkSerialmodem()
{
 CommInput pDlg;
 char buffer[30];
 DCB dcb;

 pDlg.current_baud = serial_baud_rate = 0;
 pDlg.current_com = serial_com_port = 0;

 // get the baud rate and com port to use
 while (serial_baud_rate == 0 && serial_com_port == 0)
 {
  pDlg.DoModal();

  serial_baud_rate = pDlg.current_baud;
  serial_com_port = pDlg.current_com;
  strcpy(serial_initialization_string, pDlg.m_input_init_string);
 }

 //check for the modem
 sprintf(buffer, "COM%d", serial_com_port);

 COMPortID = OpenComm(buffer, 4096, 2048);
 if (COMPortID < 0)
           MessageBox("Cannot open port", "ERROR", MB_OK);
 else
 {
  //create communication DCB
  GetCommState(COMPortID, &dcb);
//  dcb.Id = COMPortID;
  dcb.BaudRate = serial_baud_rate * 100;
  dcb.ByteSize = (BYTE)8;
  dcb.Parity = NOPARITY;
  dcb.StopBits = ONESTOPBIT;
  dcb.DsrTimeout = 1000;
  dcb.fDtrDisable = FALSE;

  if (SetCommState(&dcb) != 0)
   MessageBox("Unable to create comm structures", "ERROR", MB_OK);
  else
```

```
    {
     MessageBox("Serial Port Intialized", "Ok", MB_OK);
     current_network = NETWORK_SERIAL;
    }
  }
}

void CNetwar2View::OnUpdateNetworkSerialmodem(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_SERIAL);
}

void CNetwar2View::OnUpdateNetworkWindowsforworkgroups(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_WORKGROUPS);
}

void CNetwar2View::OnNetworkWindowsforworkgroups()
{


}

void CNetwar2View::OnNetworkWinsockinternet()
{
 WORD wVersionRequested;
 int err;

 //Start DLL find out if correct version
 wVersionRequested = 0x0101;
 err = WSAStartup(wVersionRequested, &wsaData);
 if (err != 0)
 {
  MessageBox("Winsock Version 1.1 not found...", "No Network", MB_OK | MB_ICONSTOP);
  return;
 }
 else
 {
  MessageBox("Winsock found...OK for use", "Found Network", MB_OK);
  current_network = NETWORK_INTERNET;
 }
}

void CNetwar2View::OnUpdateNetworkWinsockinternet(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(current_network == NETWORK_INTERNET);
}

void CNetwar2View::OnWaitforconnect()
{
  int len_connection_addr;
```

```
// This is for the server ONLY!!!
if (server != 0)
{
 MessageBox ("Sorry but you are not the server!!", "Not Server", MB_OK);
 return;
}

 if (current_network== NETWORK_NONE)
 {
  MessageBox("You must select a Network Type first", "Error", MB_OK);
  return;
 }

 if (total_players < 2)
 {
  MessageBox("You must select your players", "Error", MB_OK);
  return;
 }

 if (current_network == NETWORK_INTERNET)
 {
  the_server->socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (the_server->socket < 0)
 {
  MessageBox("Unable to create socket", "No Socket", MB_OK);
  return;
 }

  len_connection_addr = sizeof(struct sockaddr_in);
  the_server->connection.sin_family = AF_INET;
  the_server->connection.sin_addr.s_addr = htonl(INADDR_ANY);
  the_server->connection.sin_port = htons(5002);

  if (bind(the_server->socket,(struct sockaddr *)&the_server->connection,
len_connection_addr) < 0)
  {
   MessageBox("Unable to bind socket", "No Bind", MB_OK);
   return;
  }

  listen(the_server->socket, 3);

  //register with ASYNC
  WSAAsyncSelect(the_server->socket, m_hWnd, MSG_ACCEPT, FD_ACCEPT);
 }

 if (current_network == NETWORK_SERIAL)
 {
  if (EnableCommNotification(COMPortID, m_hWnd, 60, -1) == 0)
  {
      MessageBox("Comm notification failed", "Error", MB_OK);
      return;
    }

    //Flush both buffers
    FlushComm(COMPortID, 0);
    FlushComm(COMPortID, 1);

    //Modem Initialization String
    WriteComm(COMPortID, "AT S0=1\r", 8);
    WriteComm(COMPortID, serial_initialization_string,
strlen(serial_initialization_string));
    WriteComm(COMPortID, "\r", 1);
  }

  msg_dlg.Create();

}

long CNetwar2View::OnMsgAccept(UINT wParam, LONG lParam)
{
  int len, our_value;
  char buffer[70];

  our_value = the_server->total_connected;
  the_server->total_connected++;

  len = sizeof(struct sockaddr_in);
  the_server->clients[our_value] = accept(the_server->socket, (struct sockaddr
*)&the_server->clients_addr[our_value], &len);
  if (the_server->clients[our_value] < 0)
  {
    MessageBox("Error in Accept with client", "Error", MB_OK);
    return 0;
  }

  if (our_value == 0)
    WSAAsyncSelect(the_server->clients[our_value], m_hWnd, MSG_FROM_PLAYER1, FD_READ);
  else if (our_value == 1)
    WSAAsyncSelect(the_server->clients[our_value], m_hWnd, MSG_FROM_PLAYER2, FD_READ);
  else if (our_value == 2)
    WSAAsyncSelect(the_server->clients[our_value], m_hWnd, MSG_FROM_PLAYER3, FD_READ);

  the_server->active_connections[our_value] = TRUE;

  sprintf(buffer, "%11s %3d %3d %3d %3d %32s", "who", our_value+1, 0, 0, 0, "");
  send(the_server->clients[our_value], buffer, 60, 0);

  if (the_server->total_connected == total_players-1)
  {
   WSAAsyncSelect(the_server->socket, m_hWnd, 0, 0);

   SendInfo("game ready", 0, 0, 0, "");

   MessageBox("All Connected. Begin Game", "Go!", MB_OK);

   gold_mine_timer = SetTimer(1, 1000, NULL);
   gold_mine_rect = new CRect(185,185,235,235);

   war_button = 0;
   war_timer = SetTimer(2, 2000, NULL);

   msg_dlg.DestroyWindow();
  }

 return 1;
}

void CNetwar2View::SendInfo(char *command, int data1, int data2, int data3, char
*msg)
{
 int i;

 sprintf(send_buffer, "%11s %3d %3d %3d %3d %32s", command, data1, data2, data3,
who_am_i, msg);

 if(server == 0)
 {
  if (current_network == NETWORK_INTERNET)
  {
   for (i=1;i<total_players;i++)
   send(the_server->clients[i-1], send_buffer, 60, 0);
  }

 if (current_network == NETWORK_SERIAL)
   WriteComm(COMPortID, send_buffer, 60);
 }
 else
 {
  if (current_network == NETWORK_INTERNET)
   send(players[0]->socket, send_buffer, 60, 0);

  if (current_network == NETWORK_SERIAL)
   WriteComm(COMPortID, send_buffer, 60);
 }

  if (ACTIVATE_ACKS)
  {
   total_acks = 0;
   get_ack();
  }
```

```
}

long CNetwar2View::OnMsgFromPlayer1(UINT wParam, LONG lParam)
{
 int i;
 char buffer[70];

 i = recv(the_server->clients[0], buffer, 60, 0);
 buffer[i] = '\0';

 SendToOthers(0, buffer);

 DoCommand(buffer);
 return 1;
}


long CNetwar2View::OnMsgFromPlayer2(UINT wParam, LONG lParam)
{
 int i;
 char buffer[70];

 i = recv(the_server->clients[i], buffer, 60, 0);
 buffer[i] = '\0';

 SendToOthers(1, buffer);

 DoCommand(buffer);
 return 1;
}


long CNetwar2View::OnMsgFromPlayer3(UINT wParam, LONG lParam)
{
 int i;
 char buffer[70];

 i = recv(the_server->clients[2], buffer, 60, 0);
 buffer[i] = '\0';

 SendToOthers(2, buffer);

 DoCommand(buffer);
 return 1;
}

long CNetwar2View::SendToOthers(int client, char *buffer)
{
 int i, length;

 length = strlen(buffer);
 switch(client)
 {
   case 0: if (the_server->total_connected > 1) i = send(the_server->clients[1],
buffer, 60, 0);
         if (the_server->total_connected > 2) i = send(the_server->clients[2],
buffer, 60, 0);
         break;

   case 1: i = send(the_server->clients[0], buffer, 60, 0);
         if (the_server->total_connected > 2) i = send(the_server->clients[2],
buffer, 60, 0);
         break;

   case 2: i = send(the_server->clients[0], buffer, 60, 0);
         i = send(the_server->clients[1], buffer, 60, 0);
         break;
 }
 return 1;
}

long CNetwar2View::DoCommand(char *buffer)
{
 int player1, i, j;
 int player2;
 int sprite1;
 int sprite2;
 char command[60], msg[45];

 sscanf(buffer, "%s %d %d %d %d %s", command, &player2, &sprite1, &sprite2,
&player1, msg);

 if (strcmp(command, "who") == 0)
 {
  if (current_network != NETWORK_SERIAL)
  {
   sprintf (command, "You are connected to Server as Player #%d", player2);
   MessageBox(command, "Connected...", MB_OK);
  }
  who_am_i = player2;

  return 1;
 }
 else if (strcmp(command, "ack") == 0)
 {
  if (player2 == who_am_i)
  {
   total_acks++;
  }
  return 1;
 }
```

```
   if (ACTIVATE ACKS)
     send_ack(player1);

   if (strcmp(command, "create sprt") == 0)
   {
     for(i=0;i<sprite1;i++)
     {
       playing pieces[player1][total pieces[player1]] = new CSprite;
       switch(player2)
       {

         case 1: if (playing pieces[player1][total pieces[player1]]->Initialize(GetDC(),
bitmaps[player1][CAVALRY BITMAP MASK], bitmaps[player1][CAVALRY BITMAP],
                                            bitmaps[player1][CAVALRY BITMAP HIGHLIGHT],
                                     FALSE, 4, 3, player1, "Cavalry", 4) == FALSE)
                               MessageBox("Unable to initialize CAVALRY sprite", "Error",
MB_OK);
                            break;


         case 2: if (playing pieces[player1][total pieces[player1]]->Initialize(GetDC(),
bitmaps[player1][ARCHER BITMAP MASK], bitmaps[player1][ARCHER BITMAP],
                                       bitmaps[player1][ARCHER BITMAP HIGHLIGHT],
                                       FALSE, 4, 1, player1, "Archer", 3) == FALSE)
                               MessageBox("Unable to initialize archer sprite", "Error",
MB_OK);
                            break;



         case 3: if (playing pieces[player1][total pieces[player1]]->Initialize(GetDC(),
bitmaps[player1][LEGION_BITMAP_MASK], bitmaps[player1][LEGION BITMAP],
                                       bitmaps[player1][LEGION BITMAP HIGHLIGHT],
                                       FALSE, 8, 6, player1, "Legions", 2) == FALSE)
                               MessageBox("Unable to initialize legion sprite", "Error",
MB_OK);
                            break;

         case 4: if (playing pieces[player1][total pieces[player1]]->Initialize(GetDC(),
bitmaps[player1][FLYER BITMAP_MASK], bitmaps[player1][FLYER BITMAP],
                                            bitmaps[player1][FLYER BITMAP HIGHLIGHT],
                                     FALSE, 3, 3, player1, "Flyers", 9) == FALSE)
                               MessageBox("Unable to initialize flyer sprite", "Error",
MB OK);
                            break;
       }
       total_pieces[player1]++;
     }
   }
   else if (strcmp(command, "sprite show") == 0)
   {
```

```
  playing_pieces[player1][player2]->active = TRUE;
  Invalidate(TRUE);
}
else if (strcmp(command, "sprite_stop") == 0)
{
  playing_pieces[player1][sprite2]->MoveTo(GetDC(), player2, sprite1);
}
else if (strcmp(command, "war_button") == 0)
{
  war_button = player2;
  if (war_button == who_am_i)
  {
     ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
     war_timer = SetTimer(2, 2000, NULL);
  }
  else
   ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);
}
else if (strcmp(command, "sprite_high") == 0)
{
  playing_pieces[player2][sprite1]->war = TRUE;
  playing_pieces[player2][sprite1]->ReplaceBitmap();
  Invalidate(TRUE);
}
else if (strcmp(command, "sprite_unhi") == 0)
{
  playing_pieces[player2][sprite1]->war = FALSE;
  playing_pieces[player2][sprite1]->ReplaceBitmap();
  Invalidate(TRUE);
}
else if (strcmp(command, "sprite_offe") == 0)
{
  playing_pieces[player2][sprite1]->offense = sprite2;
  if (playing_pieces[player2][sprite1]->offense == 0 &&
     playing_pieces[player2][sprite1]->defense == 0)
  {
   playing_pieces[player2][sprite1]->active = FALSE;
   Invalidate(TRUE);
  }
}
else if (strcmp(command, "sprite_defe") == 0)
{
  playing_pieces[player2][sprite1]->defense = sprite2;
  if (playing_pieces[player2][sprite1]->offense == 0 &&
     playing_pieces[player2][sprite1]->defense == 0)
  {
   playing_pieces[player2][sprite1]->active = FALSE;
   Invalidate(TRUE);
  }
}
else if (strcmp(command, "print_str") == 0)
```

```
    {
     MessageBeep(-1);
     switch(player2)
     {
      case 1: sprintf(ops_dlg->new_message, "War: %d and %d", sprite1, sprite2);
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;

      case 2: sprintf(ops_dlg->new_message, "No Damage");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;

      case 3: sprintf(ops_dlg->new_message, "Half Damage");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;

      case 4: sprintf(ops_dlg->new_message, "Both Damage");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;

      case 5: sprintf(ops_dlg->new_message, "Attacker Damaged");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;

      case 6: sprintf(ops_dlg->new_message, "Destroyed");
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;

      case 7: sprintf(ops_dlg->new_message, "%d surrendered", sprite1);
              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
              break;
     }
    }
   else if (strcmp(command, "sprite incr") == 0)
   {
    playing_pieces[player2][sprite1]->offense = sprite2;
   }
   else if (strcmp(command, "game ready") == 0)
   {
    war_button = 0;

    ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);

    MessageBox("All Players connected. Begin Game", "Ready to go...", MB_OK);

    gold_mine_timer = SetTimer(1, 1000, NULL);
    gold_mine_rect = new CRect(185,185,235,235);
   }
   else if (strcmp(command,"msg") == 0)
   {
    if ((player2 == who_am_i) || (player2 == 4))
```

```
  {
   sprintf(ops_dlg->new_message, "%d: %s", sprite1, msg);
   ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);
  }

  MessageBeep(-1);
 }
else if (strcmp(command, "game_new") == 0)
 {
  ops_dlg->SendMessage(WM_OPERATIONS_NEW_GAME, 0, 0L);
  for (i=0;i<total_players; i++)
  {
   for (j=0;j<total_pieces[i];j++)
    delete playing_pieces[i][j];

   total_pieces[i] = 0;
  }
  players[0]->gold_coins = 500000;
  players[0]->kills = 0;
  MessageBox("Game Has Been Reset", "New Game", MB_OK);

  Invalidate(TRUE);
 }
else if (strcmp(command, "sprite_quit") == 0)
 {
  for(i=0;i<total_pieces[player2];i++)
   playing_pieces[player2][i]->active = FALSE;

  total_pieces[player2] = 0;

  Invalidate(TRUE);
 }
else if (strncmp(command, "CONNECT", 7) == 0)
 {
  if (server == 0)
  {
   SendInfo("who", 1, 0, 0, "");
   SendInfo("game_ready", 0, 0, 0, "");

   MessageBox("All Connected. Begin Game", "Go!", MB_OK);

   gold_mine_timer = SetTimer(1, 1000, NULL);


   war_button = 0;
   war_timer = SetTimer(2, 2000, NULL);

   ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
  }
 }
```

```
  return 1;
}

void CNetwar2View::OnConnecttoserver()
{
   int len_connection_addr;

   if (current_network == NETWORK_NONE)
   {
    MessageBox("You must select a Network Type first", "Error", MB_OK);
    return;
   }

   if (total_players < 2)
   {
    MessageBox("You must select your players", "Error", MB_OK);
    return;
   }

   if (server == 0)
   {
    MessageBox("You are the Server - Use Waiting For Connect..", "Error", MB_OK);
    return;
   }

   if (current_network == NETWORK_INTERNET)
   {
    if (total_players > 4)
    {
     MessageBox("Sorry but you can only have 4 players on a network connection",
"Error", MB_OK | MB_ICONSTOP);
    }
    else
    {
     len_connection_addr = sizeof(struct sockaddr_in);
     players[0]->connection.sin_family = AF_INET;
     //players[server]->get_address(address);
     players[0]->connection.sin_addr.s_addr = inet_addr(the_server->address);
     players[0]->connection.sin_port = htons(5002);
     players[0]->socket = socket(AF_INET, SOCK_STREAM, 0);

     if (players[0]->socket < 0)
     {
      MessageBox("Unable to create socket", "No Socket", MB_OK);
      return;
     }

     if (connect(players[0]->socket, (struct sockaddr *)&players[0]->connection,
len_connection_addr) < 0)
     {
        MessageBox("Unable to create a connection to server", "No Connection", MB_OK);
        return;
       }

       //register with ASYNC
       WSAAsyncSelect(players[0]->socket, m_hWnd, SERVER_MSG, FD_READ);
     }
   }

   if (current_network == NETWORK_SERIAL)
   {
     if (total_players > 2)
     {
       MessageBox("Sorry but you cannot have > 2 players on a serial connection",
"Error", MB_OK | MB_ICONSTOP);
     }
     else
     {
       //Flush both buffers
       FlushComm(COMPortID, 0);
       FlushComm(COMPortID, 1);

       if (EnableCommNotification(COMPortID, m_hWnd, 60, -1) == 0)
       {
        MessageBox("Comm notification failed", "Error", MB_OK);
        return;
       }

       if (strlen(the_server->address) > 0)
       {
        WriteComm(COMPortID, "ATDT ", 5);
        WriteComm(COMPortID, the_server->address, strlen(the_server->address));
        WriteComm(COMPortID, "\r", 1);
       }
     }
       }
}


long CNetwar2View::OnServerSend(UINT wParam, LONG lParam)
{
 char buffer[70];
 int length;

 length = recv(players[0]->socket, buffer, 60, 0);
 buffer[length] = '\0';

 DoCommand(buffer);
 return 1;
}
```

```
void CNetwar2View::OnTimer(UINT nIDEvent)
{
 int i, j;
 BOOL one_of_them, one_of_us;
 CPoint pnt;

 if ((nIDEvent == (UINT)war_timer) && (current_select == 0))
 {
  KillTimer(war_timer);

  if (who_am_i == (total_players-1))
   war_button = 0;
  else
  {
   war_button = who_am_i + 1;
   if (war_button == 1)
    lost_token_timer = SetTimer(3, 15000, NULL);
  }

  SendInfo("war_button", war_button, 0, 0, "");
  ops_dlg->SendMessage(WM_OPERATIONS_DISABLE_WAR, 0, 0L);
 }

 if (nIDEvent == (UINT)lost_token_timer)
 {
  KillTimer(lost_token_timer);

  war_button = 0;
  SendInfo("war_button", war_button, 0, 0, "");
  ops_dlg->SendMessage(WM_OPERATIONS_ENABLE_WAR, 0, 0L);
 }

 if (nIDEvent == (UINT)gold_mine_timer)
 {
  one_of_us = FALSE;
  one_of_them = FALSE;


  for(i=0;i<total_players;i++)
  {
   for(j=0;j<total_pieces[i];j++)
   {
    pnt = CPoint(playing_pieces[i][j]->mX, playing_pieces[i][j]->mY);

    if ((gold_mine_rect->PtInRect(pnt)) && (playing_pieces[i][j]->active == TRUE))
    {
     if (i == who_am_i)
      one_of_us = TRUE;
     else
      one_of_them = TRUE;
      }
    }
  }

  if ((!one_of_them) && (one_of_us))
  {
   players[0]->gold_coins += 10;
   ops_dlg->total_gold = players[0]->gold_coins;
   ops_dlg->PostMessage(WM_OPERATIONS_UPDATE_GOLD, IDOK);
   return;
  }
  }

  if (nIDEvent == (UINT)send_timer)
  {
   time_up = TRUE;
  }
}


long CNetwar2View::OnWar(UINT wParam, LONG lParam)
{
 long length;
 int i, defense, offense;
 CRect* rect;
 CPoint pnt;


 if (war_button == who_am_i)
 {

 if (current_select == 2)
 {
  current_select = 0;
  playing_pieces[second_clicked_player][second_clicked_icon]->war = FALSE;
  playing_pieces[who_am_i][first_clicked]->war = FALSE;


  // remove highlight on remote machines
  SendInfo("sprite_unhi", who_am_i, first_clicked, 0, "");
  SendInfo("sprite_unhi", second_clicked_player, second_clicked_icon, 0, "");

  // remove highlight on our machine
  playing_pieces[who_am_i][first_clicked]->ReplaceBitmap();
  playing_pieces[second_clicked_player][second_clicked_icon]->ReplaceBitmap();
  Invalidate(TRUE);


  length = (int)sqrt(pow(playing_pieces[second_clicked_player][second_clicked_icon]-
>mX-playing_pieces[who_am_i][first_clicked]->mX, 2)
```

```
                + pow(playing_pieces[second_clicked_player][second_clicked_icon]->mY-
playing_pieces[who_am_i][first_clicked]->mY, 2));

   if (length < 100)
   {
     sprintf(ops_dlg->new_message, "War - Player %d and %d", who_am_i,
second_clicked_player);
     ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0, 0L);

     //tell others to display message 1 with players who_am_i and
second_clicked_player
     SendInfo("print_str", 1, who_am_i, second_clicked_player, "");


     defense = playing_pieces[second_clicked_player][second_clicked_icon]->defense;
     offense = playing_pieces[who_am_i][first_clicked]->offense;

     switch(second_clicked_player)
     {
      case 0: rect = new CRect(0,0,70,70);
              pnt = CPoint(playing_pieces[second_clicked_player][second_
clicked_icon]->mX, playing_pieces[second_clicked_player][second_clicked_icon]->mY);

             if (rect->PtInRect(pnt))
               defense *= 2;
             break;

      case 1: rect = new CRect(380,0,480,70);
              pnt = CPoint(playing_pieces[second_clicked_player][second_
clicked_icon]->mX, playing_pieces[second_clicked_player][second_clicked_icon]->mY);

             if (rect->PtInRect(pnt))
               defense *= 2;
             break;

      case 2: rect = new CRect(0,320,70,480);
              pnt = CPoint(playing_pieces[second_clicked_player][second_
clicked_icon]->mX, playing_pieces[second_clicked_player][second_clicked_icon]->mY);

             if (rect->PtInRect(pnt))
               defense *= 2;
             break;

      case 3: rect = new CRect(380,320,480,480);
              pnt = CPoint(playing_pieces[second_clicked_player][second_
clicked_icon]->mX, playing_pieces[second_clicked_player][second_clicked_icon]->mY);

             if (rect->PtInRect(pnt))
               defense *= 2;
             break;
```

```
    }

    switch(war_matrix->get_battle_result(offense, defense))
    {
      case NO_DAMAGE: sprintf(ops_dlg->new_message, "No Damage");
                              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0,
0L);
                              SendInfo("print_str", 2, 0, 0, "");
                     break;

       case HALF_DAMAGE: sprintf(ops_dlg->new_message, "Half Damage");
                              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0,
0L);
                              SendInfo("print_str", 3, 0, 0, "");

                              if
(playing_pieces[second_clicked_player][second_clicked_icon]->offense > 0)
                              {

playing_pieces[second_clicked_player][second_clicked_icon]->offense =

playing_pieces[second_clicked_player][second_clicked_icon]->offense / 2;
                              }

                              if
(playing_pieces[second_clicked_player][second_clicked_icon]->defense > 0)
                              {

playing_pieces[second_clicked_player][second_clicked_icon]->defense =

playing_pieces[second_clicked_player][second_clicked_icon]->defense / 2;
                              }


                     SendInfo("sprite_offe", second_clicked_player,
second_clicked_icon, playing_pieces[second_clicked_player][second_clicked_icon]-
>offense, "");
                     SendInfo("sprite_defe", second_clicked_player,
second_clicked_icon, playing_pieces[second_clicked_player][second_clicked_icon]-
>defense, "");

                     if
((playing_pieces[second_clicked_player][second_clicked_icon]->defense < 0) &&

(playing_pieces[second_clicked_player][second_clicked_icon]->offense < 0))
                     {
                        playing_pieces[second_clicked_player][second_clicked_icon]-
>active = FALSE;
                        Invalidate(TRUE);
                        players[0]->kills++;
                     }
```

```
                    break;

      case ATTACKER_DAMAGE: sprintf(ops dlg->new message, "Attacker Damaged");
                              ops dlg-
>SendMessage(WM OPERATIONS UPDATE MESSAGE, 0, 0L);
                              SendInfo("print str", 5, 0, 0, "");

                          if (playing pieces[who am i][first clicked]->offense > 1)
                          {
                            playing_pieces[who am_i][first clicked]->offense =
                              playing pieces[who_am i][first_clicked]->offense /
2;
                            }


                          if (playing pieces[who am i][first clicked]->defense > 1)
                    {
                            playing pieces[who_am i][first clicked]->defense =
                              playing_pieces[who am i][first clicked]->defense /
2;
                          }

                          SendInfo("sprite offe", who am_i, first clicked,
playing pieces[who am i][first clicked]->offense, "");
                          SendInfo("sprite defe", who am i, first clicked,
playing pieces[who am i][first clicked]->defense, "");

                          if ((playing pieces[who am i][first clicked]->offense ==
0) &&
                            (playing pieces[who am_i][first clicked]->defense ==
0))
                          {
                            playing pieces[who am i][first clicked]->active =
FALSE;
                            Invalidate(TRUE);
                            SendInfo("sprite_kill", 0, 0, 0, "");
                          }
                    break;

      case BOTH DAMAGE: sprintf(ops_dlg->new message, "Both Damage");
                          ops dlg->SendMessage(WM OPERATIONS UPDATE MESSAGE, 0, 0L);
                          SendInfo("print_str", 4, 0, 0, "");

                          if
(playing pieces[second clicked player][second clicked icon]->offense > 0)
                          {
playing_pieces[second clicked_player][second clicked icon]->offense =
playing pieces[second clicked player][second clicked icon]->offense / 2;
                          }
```

```
                                  if
(playing_pieces[second_clicked_player][second_clicked_icon]->defense > 0)
                                  {

playing_pieces[second_clicked_player][second_clicked_icon]->defense =

playing_pieces[second_clicked_player][second_clicked_icon]->defense / 2;
                                  }


                        SendInfo("sprite_offe", second_clicked_player,
second_clicked_icon, playing_pieces[second_clicked_player][second_clicked_icon]-
>offense, "");
                        SendInfo("sprite_defe", second_clicked_player,
second_clicked_icon, playing_pieces[second_clicked_player][second_clicked_icon]-
>defense, "");

                        if
((playing_pieces[second_clicked_player][second_clicked_icon]->defense == 0) &&

(playing_pieces[second_clicked_player][second_clicked_icon]->offense == 0))
                        {
                           playing_pieces[second_clicked_player][second_clicked_icon]-
>active = FALSE;
                           Invalidate(TRUE);
                           players[0]->kills++;
                        }


                        if (playing_pieces[who_am_i][first_clicked]->offense > 1)
                        {
                               playing_pieces[who_am_i][first_clicked]->offense =
                               playing_pieces[who_am_i][first_clicked]->offense / 2;
                              }


                              if (playing_pieces[who_am_i][first_clicked]->defense >
1)
                        {
                               playing_pieces[who_am_i][first_clicked]->defense =
                               playing_pieces[who_am_i][first_clicked]->defense / 2;
                              }



                              SendInfo("sprite_offe", who_am_i, first_clicked, play-
ing_pieces[who_am_i][first_clicked]->offense, "");
                              SendInfo("sprite_defe", who_am_i, first_clicked, play-
ing_pieces[who_am_i][first_clicked]->defense, "");
```

```
                              if ((playing_pieces[who_am_i][first_clicked]->offense ==
0) &&
                                 (playing_pieces[who_am_i][first_clicked]->defense ==
0))
                              {
                               playing_pieces[who_am_i][first_clicked]->active =
FALSE;
                               Invalidate(TRUE);
                               SendInfo("sprite_kill", 0, 0, 0, "");
                              }
                        break;

      case DESTROYED: sprintf(ops_dlg->new_message, "Destroyed");
                              ops_dlg->SendMessage(WM_OPERATIONS_UPDATE_MESSAGE, 0,
0L);
                              SendInfo("print_str", 6, 0, 0, "");

                         playing_pieces[second_clicked_player][second_clicked_icon]-
>active = FALSE;
                         SendInfo("sprite_offe", second_clicked_player,
second_clicked_icon, 0, "");
                         SendInfo("sprite_defe", second_clicked_player,
second_clicked_icon, 0, "");
                         Invalidate(TRUE);

                         players[0]->kills++;
                         break;
    }

    if (players[0]->kills > 3)
    {
     if (playing_pieces[who_am_i][i]->offense < 12)
         playing_pieces[who_am_i][i]->offense++;

     players[0]->kills = 0;
     SendInfo("sprite_incr", who_am_i, 0, 0, "");
    }
  }
  else
  {
   MessageBox("Too far away to launch an attack", "Sorry...", MB_OK);

   playing_pieces[who_am_i][first_clicked]->war = FALSE;
   playing_pieces[second_clicked_player][second_clicked_icon]->war = FALSE;

   Invalidate(TRUE);
  }
}
}
return 1;
```

```
}

long CNetwar2View::OnMessageButton(UINT wParam, LONG lParam)
{
 CMessaging pDlg;
 char buffer[45];

 pDlg.DoModal();

 sprintf(buffer, "%s", pDlg.m_message_input);
 if (pDlg.sendto > 0)
  SendInfo("msg", pDlg.sendto-1, who_am_i, 0, buffer);

 return 1;
}

void CNetwar2View::OnOptionsExtrainvalidate()
{
  if (DO_EXTRA_INVAL)
   DO_EXTRA_INVAL = FALSE;
  else
   DO_EXTRA_INVAL = TRUE;
}

void CNetwar2View::OnUpdateOptionsExtrainvalidate(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(DO_EXTRA_INVAL == TRUE);
}

long CNetwar2View::OnCommNotify(UINT wParam, LONG lParam)
{
 char buffer[65];
 int result;

 if (wParam == (UINT)COMPortID)
 {
  switch(LOWORD(lParam))
  {
   case CN_RECEIVE: result = ReadComm(COMPortID, buffer, 60);
                    if (result < 0)
                      MessageBox("Serial Port Read Error", "Error",
MB_OK);
                    else
                    {
                      buffer[result] = '\0';
                DoCommand(buffer);
               }
               break;
  }
 }
```

```
  return 1;
 }


void CNetwar2View::OnOptionsAcknowledgements()
{
 if (ACTIVATE_ACKS)
  ACTIVATE_ACKS = FALSE;
 else
  ACTIVATE_ACKS = TRUE;
}

void CNetwar2View::OnUpdateOptionsAcknowledgements(CCmdUI* pCmdUI)
{
 pCmdUI->SetCheck(ACTIVATE_ACKS == TRUE);
}


void CNetwar2View::get_ack()

 MSG msg;
 BOOL SUCCESS;

 SUCCESS = FALSE;
 time_up = FALSE;
 while (!SUCCESS)
 {
  send_timer = SetTimer(5, 3000, NULL);
  if (current_network != NETWORK_NONE)
  {
   while ((total_acks < (total_players-1)) && (GetMessage(&msg, m_hWnd, 0, 0)))
   {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
   }

   KillTimer(send_timer);
   if ((time_up == TRUE) && (total_acks < (total_players-1)))
   {
    total_acks = 0;
    if(server == 0)
    {
     if (current_network == NETWORK_SERIAL)
      WriteComm(COMPortID, send_buffer, 60);
    }
    else
    {
     if (current_network == NETWORK_SERIAL)
      WriteComm(COMPortID, send_buffer, 60);
    }
```

```
      }
      else
      {
        SUCCESS = TRUE;
      }
    }
  }
}


void CNetwar2View::send_ack(int from)
{
  char buffer[70];
  int i;


  if(server == 0)
  {
    sprintf(buffer, "%11s %3d %3d %3d %3d %32s", "ack", from, 0, 0, 0, "");

    if (current_network == NETWORK_INTERNET)
    {
      for (i=1;i<total_players;i++)
        send(the_server->clients[i-1], send_buffer, 60, 0);
    }

    if (current_network == NETWORK_SERIAL)
      WriteComm(COMPortID, buffer, 60);
  }
  else
  {
    sprintf(buffer, "%11s %3d %3d %3d %3d %32s", "ack", from, 0, 0, who_am_i, "");

    if (current_network == NETWORK_INTERNET)
      send(players[0]->socket, buffer, 60, 0);

    if (current_network == NETWORK_SERIAL)
      WriteComm(COMPortID, buffer, 60);
  }
}
```

ommiput.h

```
// omminput.h : header file
//

/////////////////////////////////////////////////////////////////////////////
```

```
// CommInput dialog

class CommInput : public CDialog
{
// Construction
public:
	CommInput(CWnd* pParent = NULL);  // standard constructor
	int current_baud;
	int current_com;

// Dialog Data
	//{{AFX_DATA(CommInput)
	enum { IDD = IDD_COMM_INPUT };
	CString	m_input_init_string;
	//}}AFX_DATA

// Implementation
protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV support

	// Generated message map functions
	//{{AFX_MSG(CommInput)
	afx_msg void OnBaud19200();
	afx_msg void OnBaud2400();
	afx_msg void OnBaud38400();
	afx_msg void OnBaud4800();
	afx_msg void OnBaud57600();
	afx_msg void OnBaud9600();
	afx_msg void OnCom1();
	afx_msg void OnCom2();
	afx_msg void OnCom3();
	afx_msg void OnCom4();
	afx_msg void OnOK();
	afx_msg void OnCancel();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

omminput.cpp

```
// omminput.cpp : implementation file
//

#include "stdafx.h"
#include "netwar2.h"
#include "omminput.h"
```

```
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CommInput dialog


CommInput::CommInput(CWnd* pParent /*=NULL*/)
        : CDialog(CommInput::IDD, pParent)
{
        //{{AFX_DATA_INIT(CommInput)
        m_input_init_string = "";
        //}}AFX_DATA_INIT
}

void CommInput::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CommInput)
        DDX_Text(pDX, IDC_INPUT_INITIALIZATION_STRING, m_input_init_string);
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CommInput, CDialog)
        //{{AFX_MSG_MAP(CommInput)
        ON_BN_CLICKED(IDC_BAUD_19200, OnBaud19200)
        ON_BN_CLICKED(IDC_BAUD_2400, OnBaud2400)
        ON_BN_CLICKED(IDC_BAUD_38400, OnBaud38400)
        ON_BN_CLICKED(IDC_BAUD_4800, OnBaud4800)
        ON_BN_CLICKED(IDC_BAUD_57600, OnBaud57600)
        ON_BN_CLICKED(IDC_BAUD_9600, OnBaud9600)
        ON_BN_CLICKED(IDC_COM_1, OnCom1)
        ON_BN_CLICKED(IDC_COM_2, OnCom2)
        ON_BN_CLICKED(IDC_COM_3, OnCom3)
        ON_BN_CLICKED(IDC_COM_4, OnCom4)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// CommInput message handlers

void CommInput::OnBaud19200()
{
  current_baud = 192;
}

void CommInput::OnBaud2400()
{
 current_baud = 24;
}

void CommInput::OnBaud38400()
{
 current_baud = 384;
}

void CommInput::OnBaud4800()
{
 current_baud = 48;
}

void CommInput::OnBaud57600()
{
 current_baud = 576;
}

void CommInput::OnBaud9600()
{
 current_baud = 96;
}

void CommInput::OnCom1()
{
 current_com = 1;
}

void CommInput::OnCom2()
{
 current_com = 2;
}

void CommInput::OnCom3()
{
 current_com = 3;
}

void CommInput::OnCom4()
{
 current_com = 4;
}

void CommInput::OnOK()
{
	CDialog::OnOK();
}

void CommInput::OnCancel()
{
 CDialog::OnCancel();
}
```

operatio.h

```
// operatio.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// operations dialog


#define WM_OPERATIONS_ADD_ARMIES    WM_USER + 100
#define WM_OPERATIONS_UPDATE_GOLD   WM_USER + 101
#define WM_OPERATIONS_ENABLE_WAR    WM_USER + 102
#define WM_OPERATIONS_DISABLE_WAR   WM_USER + 103
#define WM_OPERATIONS_UPDATE_MESSAGE WM_USER + 104
#define WM_OPERATIONS_NEW_GAME       WM_USER + 105
#define WM_MESSAGE_BUTTON            WM_USER + 106
#define WM_OPERATIONS_PLAYER_QUITS   WM_USER + 107


class operations : public CDialog
{
// Construction
public:
    operations(CWnd* parent = NULL);
        operations(CView* pView); // standard constructor
        BOOL Create();
        CView* m_pView;

        int total_armies;
        char entries[50][20];
        int list_box[25];
        int selected_army;

        long total_gold;

        int total_messages;
        char new_message[50];

// Dialog Data
        //{{AFX_DATA(operations)
        enum { IDD = IDD_OPERATIONS_DIALOG };
        CString  m_operations_available_gold;
        CString  m_operations_war_statement;
        //}}AFX_DATA

// Implementation
protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        // Generated message map functions
        //{{AFX_MSG(operations)
        afx_msg void OnWarButton();
        afx_msg void OnOperationsIQuit();
        afx_msg void OnOperationsMessageButton();
        afx_msg void OnDblclkAvailableArmies();
        //}}AFX_MSG
        long OnAddArmies(UINT wParam, LONG lParam);
        long OnUpdateGold(UINT wParam, LONG lParam);
        long OnWarDo(UINT wParam, LONG lParam);
        long OnWarUndo(UINT wParam, LONG lParam);
        long OnUpdateMessage(UINT wParam, LONG lParam);
        long OnNewGame(UINT wParam, LONG lParam);
        DECLARE_MESSAGE_MAP()
};
```

operatio.cpp

```
// operatio.cpp : implementation file
//

#include "stdafx.h"
#include "netwar2.h"
#include "operatio.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// operations dialog


operations::operations(CWnd* pParent /*=NULL*/)
        : CDialog(operations::IDD, pParent)
{
        //{{AFX_DATA_INIT(operations)
        m_operations_available_gold = "";
        m_operations_war_statement = "";
        //}}AFX_DATA_INIT

    m_pView = NULL;

        total_armies = 0;
        total_messages = 0;
}
```

```
operations::operations(CView* pView) // modeless constructor
    : CDialog()
{
  m_operations_available_gold = "";
  m_operations_war_statement = "";

  total_armies = 0;
  total_gold = 0;

  m_pView = pView;
}
BOOL operations::Create()
{
  return CDialog::Create(operations::IDD);
}

void operations::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(operations)
        DDX_Text(pDX, IDC_OPERATIONS_AVAILABLE_GOLD, m_operations_available_gold);
        DDX_Text(pDX, IDC_OPERATIONS_WAR_STATEMENT, m_operations_war_statement);
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(operations, CDialog)
        ON_MESSAGE(WM_OPERATIONS_ADD_ARMIES, OnAddArmies)
        ON_MESSAGE(WM_OPERATIONS_UPDATE_GOLD, OnUpdateGold)
        ON_MESSAGE(WM_OPERATIONS_ENABLE_WAR, OnWarDo)
        ON_MESSAGE(WM_OPERATIONS_DISABLE_WAR, OnWarUndo)
        ON_MESSAGE(WM_OPERATIONS_UPDATE_MESSAGE, OnUpdateMessage)
        ON_MESSAGE(WM_OPERATIONS_NEW_GAME, OnNewGame)
        //{{AFX_MSG_MAP(operations)
        ON_BN_CLICKED(IDC_WAR_BUTTON, OnWarButton)
        ON_BN_CLICKED(IDC_OPERATIONS_1_QUIT, OnOperationsIQuit)
        ON_BN_CLICKED(IDC_OPERATIONS_MESSAGE_BUTTON, OnOperationsMessageButton)
        ON_LBN_DBLCLK(IDC_AVAILABLE_ARMIES, OnDblclkAvailableArmies)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// operations message handlers

void operations::OnWarButton()
{
        if (m_pView != NULL)
        {
          m_pView->SendMessage(WM_WAR_BUTTON, IDOK);
        }
}
```

```
void operations::OnOperationsIQuit()
{
 if (m_pView != NULL)
  m_pView->SendMessage(WM_QUIT_BUTTON, IDOK);
}

void operations::OnOperationsMessageButton()
{
 if (m_pView != NULL)
  m_pView->SendMessage(WM_MESSAGE_BUTTON, IDOK);
}

void operations::OnDblclkAvailableArmies()
{
 CListBox* box;
 int temp;

 box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
 temp = box->GetCurSel();

 if (list_box[temp] != -1)
 {
  selected_army = list_box[temp];
  list_box[temp] = -1;

  box->DeleteString(temp);
  box->InsertString(temp, "Army Deployed");

  if (m_pView != NULL)
  m_pView->SendMessage(WM_ARMY_SELECTED, IDOK);
 }
}

long operations::OnAddArmies(UINT wParam, LONG lParam)
{
 int i;
 CListBox* box;

 box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
 box->ResetContent();
 for(i=0;i<total_armies;i++)
 {
  if (list_box[i] == -1)
   box->AddString("Army Deployed");
  else
   box->AddString(entries[i]);
 }

 return 1;
}
```

```
long operations::OnUpdateGold(UINT wParam, LONG lParam)
{
   CEdit* box;
   char buffer[15];

   box = (CEdit*)GetDlgItem(IDC_OPERATIONS_AVAILABLE_GOLD);
   sprintf(buffer, "%ld", total_gold);
   box->SetSel(0, -1, FALSE);
   box->ReplaceSel(buffer);

   return 1;
}

long operations::OnWarDo(UINT wParam, LONG lParam)
{
   CEdit* box;

   box = (CEdit*)GetDlgItem(IDC_OPERATIONS_WAR_STATEMENT);
   box->SetSel(0, -1, FALSE);
   box->ReplaceSel("Okay For");

  return 1;
}

long operations::OnWarUndo(UINT wParam, LONG lParam)
{
   CEdit* box;

   box = (CEdit*)GetDlgItem(IDC_OPERATIONS_WAR_STATEMENT);
   box->SetSel(0, -1, FALSE);
   box->ReplaceSel("Not Okay");

   return 1;
}

long operations::OnUpdateMessage(UINT wParam, LONG lParam)
{
   CListBox* box;

   box = (CListBox*)GetDlgItem(IDC_OPERATIONS_MESSAGE_BOX);

   box->SetHorizontalExtent(256);

   total_messages++;
   if (total_messages > 10)
   {
    box->DeleteString(0);
    box->AddString(new_message);
   }
```

```
    else
    {
      box->AddString(new_message);
    }

    return 1;
}

long operations::OnNewGame(UINT wParam, LONG lParam)
{
  CListBox* box;
  CEdit* box2;

  box = (CListBox*)GetDlgItem(IDC_AVAILABLE_ARMIES);
  box->ResetContent();

  box = (CListBox*)GetDlgItem(IDC_OPERATIONS_MESSAGE_BOX);
  box->ResetContent();

  box2 = (CEdit*)GetDlgItem(IDC_OPERATIONS_AVAILABLE_GOLD);
  box2->SetSel(0, -1, FALSE);
  box2->ReplaceSel("");


  total_armies = 0;
  return 1;
}
```

player.h

```
#include "winsock.h"


class CPlayer
{
public:
      long gold_coins;
      int kills;
      int socket;
      struct sockaddr_in connection;

public:
         CPlayer();
         ~CPlayer();
};
```

player.cpp

```
#include "player.h"
#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
#include "netwavw.h"

CPlayer::CPlayer()
{
   gold_coins = 500000;
   kills = 0;
}

CPlayer::~CPlayer()
{
}
```

ppstats.h

```
// ppstats.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// PPSTATS dialog

class PPSTATS : public CDialog
{
// Construction
public:
        PPSTATS(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
        //{{AFX_DATA(PPSTATS)
        enum { IDD = IDD_PLAYING_PIECE_STATS };
        CString m_stats_defense;
        CString m_stats_offense;
        CString m_stats_owner;
        CString m_stats_type;
        //}}AFX_DATA

// Implementation
protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
```

```
	// Generated message map functions
	//{{AFX_MSG(PPSTATS)
		// NOTE: the ClassWizard will add member functions here
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

ppstats.cpp

```
// ppstats.cpp : implementation file
//

#include "stdafx.h"
#include "netwar2.h"
#include "ppstats.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// PPSTATS dialog


PPSTATS::PPSTATS(CWnd* pParent /*=NULL*/)
	: CDialog(PPSTATS::IDD, pParent)
{
	//{{AFX_DATA_INIT(PPSTATS)
	m_stats_defense = "";
	m_stats_offense = "";
	m_stats_owner = "";
	m_stats_type = "";
	//}}AFX_DATA_INIT
}

void PPSTATS::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(PPSTATS)
	DDX_Text(pDX, IDC_STATS_DEFENSE, m_stats_defense);
	DDX_Text(pDX, IDC_STATS_OFFENSE, m_stats_offense);
	DDX_Text(pDX, IDC_STATS_OWNER, m_stats_owner);
	DDX_Text(pDX, IDC_STATS_TYPE, m_stats_type);
	//}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(PPSTATS, CDialog)
        //{{AFX_MSG_MAP(PPSTATS)
                // NOTE: the ClassWizard will add message map macros here
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// PPSTATS message handlers
```

resource.h

```
//{{NO_DEPENDENCIES}}
// App Studio generated include file.
// Used by NETWAR2.RC
//
#define IDR_MAINFRAME                           2
#define IDD_ABOUTBOX                            100
#define IDB_PLAYING_FIELD                       102
#define IDB_CAVALRY_BITMAP                      103
#define IDB_CAVALRY_BITMAP_MASK                 104
#define IDD_PLAYING_PIECE_STATS                 104
#define IDB_CAVALRY_BITMAP_HIGHLIGHT            105
#define IDB_CAVALRY_BITMAP_HLIGHT               105
#define IDB_CAVALRY_BITMAP_HL                   105
#define IDD_OPERATIONS_DIALOG                   105
#define IDD_SERVER_WAIT                         106
#define IDD_INPUT_PLAYER                        109
#define IDD_INPUT_SERVER                        109
#define IDD_INPUT_NAME                          110
#define IDD_INPUT_SELF                          110
#define IDD_BUY_EQUIPMENT                       114
#define IDB_LEGION_PLAYER_0                     127
#define IDB_LEGION_BITMAP                       127
#define IDD_MESSAGE_INPUT                       129
#define IDD_COMM_INPUT                          130
#define IDB_ARCHER_PLAYER_0                     131
#define IDB_ARCHER_BITMAP                       131
#define IDB_FLYING_PLAYER_0                     135
#define IDB_FLYING_BITMAP                       135
#define IDB_FLYER_BITMAP                        135
#define IDB_LEGION_PLAYER_0_h                   143
#define IDB_LEGION_BITMAP_HL                    143
#define IDB_NEW_FLYER_0                         145
#define IDB_PO_FLYER_BITMAP                     145
#define IDB_NEW_CAVALRY_0                       146
#define IDB_PO_CAVALRY_BITMAP                   146
```

```
#define IDB_FLYING_PLAYER_0_h           147
#define IDB_FLYING_BITMAP_HL            147
#define IDB_FLYER_BITMAP_HL             147
#define IDB_ARCHER_PLAYER_0_h           151
#define IDB_ARCHER_BITMAP_HL            151
#define IDB_ARCHER_PLAYER_0_M           155
#define IDB_ARCHER_BITMAP_MASK          155
#define IDB_NEW_ARCHER_0                164
#define IDB_P0_ARCHER_BITMAP            164
#define IDB_NEW_LEGION_0                165
#define IDB_P0_LEGION_BITMAP            165
#define IDB_FLYING_PLAYER_0_m           177
#define IDB_FLYING_BITMAP_MASK          177
#define IDB_FLYER_BITMAP_MASK           177
#define IDB_LEGION_PLAYER_0_m           185
#define IDB_LEGION_BITMAP_MASK          185
#define IDB_NEW_FLYER_0_MASK            193
#define IDB_P0_FLYER_MASK               193
#define IDB_NEW_FLYER_0_H               194
#define IDB_P0_FLYER_HL                 194
#define IDB_NEW_CAVALRY_0_H             195
#define IDB_P0_CAVALRY_HL               195
#define IDB_NEW_ARCHER_0_H              196
#define IDB_P0_ARCHER_HL                196
#define IDB_NEW_LEGION_0_H              197
#define IDB_P0_LEGION_HL                197
#define IDB_P0_ARCHER_MASK              198
#define IDB_P0_CAVALRY_MASK             199
#define IDB_P0_LEGION_MASK              200
#define ADD_ARMY_SOUND                  201
#define IDB_P1_ARCHER_BITMAP            201
#define IDB_P1_ARCHER_HL                202
#define IDB_P1_ARCHER_MASK              203
#define IDB_P1_CAVALRY_BITMAP           204
#define IDB_P1_CAVALRY_HL               205
#define IDB_P1_CAVALRY_MASK             206
#define IDB_P1_FLYER_BITMAP             207
#define IDB_P1_FLYER_HL                 208
#define IDB_P1_FLYER_MASK               209
#define IDB_P1_LEGION_BITMAP            211
#define IDB_P1_LEGION_HL                212
#define IDB_P1_LEGION_MASK              213
#define IDB_P2_ARCHER_BITMAP            216
#define IDB_P2_ARCHER_HL                217
#define IDB_P2_ARCHER_MASK              218
#define IDB_P2_CAVALRY_BITMAP           219
#define IDB_P2_CAVALRY_HL               220
#define IDB_P2_CAVALRY_MASK             221
#define IDB_P2_FLYER_BITMAP             222
#define IDB_P2_FLYER_HL                 223
```

```
#define IDB_P2_FLYER_MASK                         224
#define IDB_P2_LEGION_BITMAP                      225
#define IDB_P2_LEGION_HL                          226
#define IDB_P2_LEGION_MASK                        227
#define IDB_P3_ARCHER_BITMAP                      228
#define IDB_P3_ARCHER_HL                          229
#define IDB_P3_ARCHER_MASK                        230
#define IDB_P3_CAVALRY_BITMAP                     231
#define IDB_P3_CAVALRY_HL                         232
#define IDB_P3_CAVALRY_MASK                       233
#define IDB_P3_FLYER_BITMAP                       234
#define IDB_P3_FLYER_HL                           235
#define IDB_P3_FLYER_MASK                         236
#define IDB_P3_LEGION_BITMAP                      237
#define IDB_P3_LEGION_HL                          238
#define IDB_P3_LEGION_MASK                        239
#define IDC_STATS_TYPE                            1000
#define IDC_AVAILABLE_ARMIES                      1001
#define Gold                                      1002
#define IDC_OPERATIONS_AVAILABLE_GOLD             1003
#define IDC_STATS_OFFENSE                         1004
#define IDC_OPERATIONS_WAR_STATEMENT              1004
#define IDC_STATS_OWNER                           1005
#define IDC_WAR_BUTTON                            1005
#define IDC_STATS_DEFENSE                         1006
#define IDC_OPERATIONS_I_QUIT                     1006
#define IDC_OPERATIONS_MESSAGE_BUTTON             1007
#define IDC_OPERATIONS_MESSAGE_BOX                1008
#define IDC_EDIT2                                 1008
#define IDC_INPUT_PLAYER_INTERNET                 1009
#define ID_OK                                     1010
#define IDC_RADIO_CLIENT                          1021
#define IDC_RADIO_SERVER                          1022
#define IDC_INPUT_TOTAL_PLAYERS                   1028
#define IDC_BUY_CAVALRYS                          1031
#define IDC_BUY_ARCHERS                           1032
#define IDC_SELL_ARCHERS                          1033
#define IDC_SELL_CAVALRYS                         1034
#define IDC_BUY_LEGION                            1035
#define IDC_SELL_LEGION                           1036
#define IDC_BUY_CAVALRYS_COUNT                    1037
#define IDC_BUY_ARCHERS_COUNT                     1038
#define IDC_BUY_AVAILABLE_COINS                   1039
#define IDC_SELL_FLYERS                           1040
#define IDC_BUY_FLYERS                            1041
#define IDC_BUY_LEGIONS_COUNT                     1052
#define IDC_BUY_FLYERS_COUNT                      1053
#define IDC_MESSAGE_INPUT                         1055
#define IDC_SEND_MESSAGE_1                        1056
#define IDC_SEND_MESSAGE_2                        1057
#define IDC_COM_1                                 1057
```

```
#define IDC_SEND_MESSAGE_3                   1058
#define IDC_COM_2                            1058
#define IDC_SEND_MESSAGE_4                   1059
#define IDC_COM_3                            1059
#define IDC_SEND_MESSAGE_ALL                 1060
#define IDC_COM_4                            1060
#define IDC_BAUD_2400                        1061
#define IDC_INPUT_INITIALIZATION_STRING      1062
#define IDC_BAUD_4800                        1063
#define IDC_BAUD_9600                        1064
#define IDC_BAUD_19200                       1065
#define IDC_BAUD_38400                       1066
#define IDC_BAUD_57600                       1067
#define ID_PLAYERS_BUYEQUIPMENT              32771
#define ID_OPTIONS_SOUND                     32772
#define ID_PLAYERS_INPUTPLAYERS              32773
#define ID_NETWORK_WINSOCKINTERNET           32774
#define ID_NETWORK_SERIALMODEM               32775
#define ID_NETWORK_WINDOWSFORWORKGROUPS      32776
#define ID_WAITFORCONNECT                    32777
#define ID_CONNECTTOSERVER                   32778
#define ID_OPTIONS_EXTRAINVALIDATE           32779
#define ID_OPTIONS_ACKNOWLEDGEMENTS          32780

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS

#define _APS_NEXT_RESOURCE_VALUE             107
#define _APS_NEXT_COMMAND_VALUE              32781
#define _APS_NEXT_CONTROL_VALUE              1009
#define _APS_NEXT_SYMED_VALUE                102
#endif
#endif
```

server.h

```
#include "winsock.h"


class CServer
{
 public:
   int socket;
   struct sockaddr_in connection;

   BOOL active_connections[3]; // true if connection good
   int total_connected;            // total players connected to the server
   int clients[3];                     // sockets for connected players
   struct sockaddr_in clients_addr[3]; // structures for connected players
   char address[25];                 // Internet address or telephone number

  public:
   CServer();
   ~CServer();
};
```

server.cpp

```
#include "server.h"
#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
#include "netwavw.h"

CServer::CServer()
{
  int i;
  total_connected = 0;
  for(i=0;i<3;i++)
    active_connections[i] = FALSE;

}

CServer::~CServer()
{
}
```

serverwa.h

```
// serverwa.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CServerWait dialog

class CServerWait : public CDialog
{
// Construction
public:
        CServerWait(CWnd* pParent = NULL);   // standard constructor
        Create();

// Dialog Data
        //{{AFX_DATA(CServerWait)
        enum { IDD = IDD_SERVER_WAIT };
                // NOTE: the ClassWizard will add data members here
        //}}AFX_DATA

// Implementation
protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

        // Generated message map functions
        //{{AFX_MSG(CServerWait)
                // NOTE: the ClassWizard will add member functions here
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

serverwa.cpp

```
// serverwa.cpp : implementation file
//

#include "stdafx.h"
#include "netwar2.h"
#include "serverwa.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CServerWait dialog


CServerWait::CServerWait(CWnd* pParent /*=NULL*/)
        : CDialog(CServerWait::IDD, pParent)
{
        //{{AFX_DATA_INIT(CServerWait)
                // NOTE: the ClassWizard will add member initialization here
        //}}AFX_DATA_INIT
}

CServerWait::Create()
{
 return CDialog::Create(CServerWait::IDD);
}

void CServerWait::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CServerWait)
		// NOTE: the ClassWizard will add DDX and DDV calls here
 //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CServerWait, CDialog)
	//{{AFX_MSG_MAP(CServerWait)
		// NOTE: the ClassWizard will add message map macros here
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CServerWait message handlers
```

sprite.h

```
/////////////////////////////////////////////////////////////////////////////
//                                                                         //
// SPRITE.H: Header file for the CSprite class.                            //
//                                                                         //
/////////////////////////////////////////////////////////////////////////////
#include <time.h>

class CSprite
{
public:
 int XOffset, YOffset,
     mHeight, mWidth,
     mX, mY;
 BOOL show;       // If TRUE then we have shown this sprite
 BOOL active;
 BOOL war;
 int defense,
     offense,
     player,
     speed,
     startx,
     starty;

 char type[10];
private:
  CBitmap* mHImage;
  CBitmap* mHMask;
  CBitmap* mHSave;
  CBitmap* mHHighlight;
  CBitmap* mHHighlightmask;
  CBitmap* mtemp;
  CBitmap* mtempmask;

  BOOL switched;


public:
  CSprite ();
  ~CSprite ();

  void GetCoord (CRect *Rect);
  BOOL Hide (CDC* HDc);
  BOOL Initialize (CDC* Hdc,
                              CBitmap* HMask,
                              CBitmap* HImage,
                              CBitmap* HHighlight,
                              BOOL Active,
                              int Defense,
                              int Offense,
                              int Player,
                              char *Type,
                              int Speed
                              );
  BOOL MoveTo (CDC* HDc, int X, int Y);
  BOOL Redraw (CDC* HDc);
  BOOL Start (CDC* HDc, int X, int Y);
  BOOL ReplaceBitmap();

};
```

sprite.cpp

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
// SPRITE.CPP: Member functions for the CSprite class.                        //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
```

```
#include "netwavw.h"
#include <time.h>

CSprite::CSprite ()
{
 mHImage = 0;
 mHMask = 0;
 mHSave = 0;
 mHHighlight = 0;
 mHHighlightmask = 0;
 mX = mY = 0;
 mWidth = mHeight = 0;
 switched = FALSE;
 active = FALSE;
 war = FALSE;

 defense = 0;
 offense = 0;
 player = -1;
 speed = 0;
}

CSprite::~CSprite ()
 {
 if (mHSave != 0)
    delete (mHSave);
 }

void CSprite::GetCoord (CRect *Rect)
{
 Rect->left = mX;
 Rect->top = mY;
 Rect->right = mX + mWidth;
 Rect->bottom = mY + mHeight;
}

BOOL CSprite::Hide (CDC* HDc)
{
 CDC* HMemDC;

 if (mHSave == 0)
    return (FALSE);

 HMemDC = new CDC;
 HMemDC->CreateCompatibleDC (HDc);
 if (HMemDC == NULL)
 {
  delete HMemDC;
  return (FALSE);
 }
```

```
  // restore window graphics at prior position:
  HMemDC->SelectObject (mHSave);
  if (HDc->BitBlt(mX, mY, mWidth, mHeight, HMemDC, 0, 0, SRCCOPY) == FALSE)
  {
   delete HMemDC;
   return(FALSE);
  }

  delete HMemDC;
  return (TRUE);
}


BOOL CSprite::ReplaceBitmap()
{
  if (switched == FALSE)
  {
   mtemp = mHImage;
   mHImage = mHHighlight;
   switched = TRUE;
  }
  else
  {
   mHImage = mtemp;
   switched = FALSE;
  }

  return TRUE;
}


BOOL CSprite::Initialize (CDC* Hdc,
                          CBitmap* HMask,
                          CBitmap* HImage,
                          CBitmap* HHighlight,
                          BOOL Active,
                          int Defense,
                          int Offense,
                          int Player,
                          char *Type,
                          int Speed
                          )
{
 CBitmap* HSave;
 BITMAP ImageBM;
 BITMAP MaskBM;
 BITMAP HighlightBM;
 int Result;

 Result = HMask->GetObject (sizeof (BITMAP), &MaskBM);
 if (Result == 0)
  return (FALSE);

 Result = HImage->GetObject (sizeof (BITMAP), &ImageBM);
 if (Result == 0)
  return (FALSE);

 Result = HHighlight->GetObject(sizeof(BITMAP), &HighlightBM);
 if (Result == 0)
  return (FALSE);


 if (MaskBM.bmWidth != ImageBM.bmWidth ||
 MaskBM.bmHeight != ImageBM.bmHeight)
  return (FALSE);

 HSave = new CBitmap;
 HSave->CreateCompatibleBitmap(Hdc, MaskBM.bmWidth, MaskBM.bmHeight);

 if (HSave == NULL)
 {
  delete HSave;
  return (FALSE);
 }

 if (mHSave != 0)
  delete mHSave;

 mHSave = HSave;
 mHMask = HMask;
 mHImage = HImage;
 mHHighlight = HHighlight;
 mWidth = MaskBM.bmWidth;
 mHeight = MaskBM.bmHeight;

 defense = Defense;
 offense = Offense;
 player = Player;
 active = Active;
 speed = Speed;
 strcpy(type, Type);


 show = FALSE;

 return (TRUE);
}

BOOL CSprite::MoveTo (CDC* HDc, int X, int Y)
{
 CBitmap* HBMHold;
 CDC* HMemDC;
 CDC* HMemDCHold;
 RECT RectNew;
 RECT RectOld;
 RECT RectUnion;

 // copy entire affected area of window to working bitmap:
 RectOld.left = mX;
 RectOld.top = mY;
 RectOld.right = mX + mWidth;
 RectOld.bottom = mY + mHeight;

 RectNew.left = X;
 RectNew.top = Y;
 RectNew.right = X + mWidth;
 RectNew.bottom = Y + mHeight;

 UnionRect (&RectUnion, &RectOld, &RectNew);
 RectUnion.left -= RectUnion.left % 8;

 HBMHold = new CBitmap;
 if (HBMHold->CreateCompatibleBitmap(HDc, RectUnion.right - RectUnion.left,
                                      RectUnion.bottom - RectUnion.top) == NULL)
 {
  delete HBMHold;
  return(FALSE);
 }

 HMemDCHold = new CDC;
 HMemDCHold->CreateCompatibleDC (HDc);
 HMemDCHold->SelectObject (HBMHold);

 HMemDCHold->BitBlt(0, 0, RectUnion.right - RectUnion.left,
                         RectUnion.bottom - RectUnion.top,
                         HDc, RectUnion.left, RectUnion.top, SRCCOPY);

 // create another memory device context:
 HMemDC = new CDC;
 HMemDC->CreateCompatibleDC(HDc);

 // restore window graphics at prior position:
 HMemDC->SelectObject (mHSave);
 HMemDCHold->BitBlt(mX - RectUnion.left, mY - RectUnion.top, mWidth, mHeight,
                  HMemDC, 0, 0, SRCCOPY);

 // save current window graphics:
 HMemDC->BitBlt(0, 0, mWidth, mHeight, HMemDCHold, X - RectUnion.left, Y -
RectUnion.top,
                  SRCCOPY);

 // display mask bitmap:
```

```
 HMemDC->SelectObject (mHMask);
 HMemDCHold->BitBlt(X - RectUnion.left, Y - RectUnion.top, mWidth, mHeight,
                    HMemDC, 0, 0, SRCAND);

 // display image bitmap:
 HMemDC->SelectObject (mHImage);
 HMemDCHold->BitBlt(X - RectUnion.left, Y - RectUnion.top, mWidth, mHeight,
                    HMemDC, 0, 0, SRCINVERT);

 // copy working bitmap back to window:
 HDc->BitBlt(RectUnion.left, RectUnion.top, RectUnion.right - RectUnion.left,
             RectUnion.bottom - RectUnion.top, HMemDCHold, 0, 0, SRCCOPY);

 // delete the memory device contexts:
 delete HMemDCHold;
 delete HMemDC;

 // delete working bitmap:
 delete HBMHold;

 mX = X;
 mY = Y;

 return (TRUE);
}

BOOL CSprite::Redraw (CDC* HDc)
{
 CDC* HMemDC;

 if (mHSave == 0)
    return (FALSE);

 HMemDC = new CDC;
 if (HMemDC->CreateCompatibleDC(HDc)==FALSE)
 {
  delete HMemDC;
  return (FALSE);
 }

 // display mask bitmap:
 HMemDC->SelectObject(mHMask);
 if (HDc->BitBlt (mX, mY, mWidth, mHeight, HMemDC, 0, 0, SRCAND) == FALSE)
 {
  delete HMemDC;
  return (FALSE);
 }

 // display image bitmap:
 HMemDC->SelectObject (mHImage);
 if (HDc->BitBlt(mX, mY, mWidth, mHeight, HMemDC, 0, 0, SRCINVERT) == FALSE)
```

```
  {
   delete HMemDC;
   return(FALSE);
  }

 delete HMemDC;
 return (TRUE);
}

BOOL CSprite::Start (CDC* HDc, int X, int Y)
{
 CDC* HMemDC;

 if (mHSave == 0)
  return (FALSE);

 HMemDC = new CDC;
 if (HMemDC->CreateCompatibleDC (HDc) == FALSE)
 {
  delete HMemDC;
  return (FALSE);
 }

 // save current window graphics:
 HMemDC->SelectObject (mHSave);
 if (HMemDC->BitBlt(0, 0, mWidth, mHeight, HDc, X, Y, SRCCOPY) == FALSE)
 {
  delete HMemDC;
  return (FALSE);
 }

 // display mask bitmap:
 HMemDC->SelectObject (mHMask);
 if (HDc->BitBlt(X, Y, mWidth, mHeight, HMemDC, 0, 0, SRCAND) == FALSE)
 {
  delete HMemDC;
  return (FALSE);
 }

 // display image bitmap:
 HMemDC->SelectObject (mHImage);
 if (HDc->BitBlt(X, Y, mWidth, mHeight, HMemDC, 0, 0, SRCINVERT) == FALSE)
 {
  delete HMemDC;
  return(FALSE);
 }

 delete HMemDC;
 mX = X;
 mY = Y;
 return (TRUE);
 }
```

warmatrx.h

```
#define NO_DAMAGE 0
#define HALF_DAMAGE 1
#define BOTH_DAMAGE 2
#define ATTACKER_DAMAGE 3
#define DESTROYED 4


class CBattle
{
 public:
       get_battle_result(int offense, int defense);

 private:
    int battle_matrix[12][12][6];

    public:
    CBattle();
    ~CBattle();
};
```

warmatrx.cpp

```
#include "stdafx.h"
#include "netwar2.h"
#include "netwadoc.h"
#include "netwavw.h"
#include <time.h>

CBattle::CBattle()
{
   int i, j, k;

   srand((unsigned) time(NULL));
   //zero out battle matrix
   for(i=0;i<12;i++)
    for(j=0;j<12;j++)
     for(k=0;k<6;k++)
      battle_matrix[i][j][k] = NO_DAMAGE;
```

```
 //0 to 0
 battle_matrix[0][0][0] = NO_DAMAGE;
 battle_matrix[0][0][1] = BOTH_DAMAGE;
 battle_matrix[0][0][2] = NO_DAMAGE;
 battle_matrix[0][0][3] = BOTH_DAMAGE;
 battle_matrix[0][0][4] = NO_DAMAGE;
 battle_matrix[0][0][5] = NO_DAMAGE;

 //0 to 1
 battle_matrix[0][1][0] = NO_DAMAGE;
 battle_matrix[0][1][1] = BOTH_DAMAGE;
 battle_matrix[0][1][2] = ATTACKER_DAMAGE;
 battle_matrix[0][1][3] = NO_DAMAGE;
 battle_matrix[0][1][4] = NO_DAMAGE;
 battle_matrix[0][1][5] = ATTACKER_DAMAGE;

 //0 to 2
 battle_matrix[0][2][0] = BOTH_DAMAGE;
 battle_matrix[0][2][1] = NO_DAMAGE;
 battle_matrix[0][2][2] = ATTACKER_DAMAGE;
 battle_matrix[0][2][3] = ATTACKER_DAMAGE;
 battle_matrix[0][2][4] = NO_DAMAGE;
 battle_matrix[0][2][5] = ATTACKER_DAMAGE;

 //0 to 3
 battle_matrix[0][3][0] = NO_DAMAGE;
 battle_matrix[0][3][1] = NO_DAMAGE;
 battle_matrix[0][3][2] = ATTACKER_DAMAGE;
 battle_matrix[0][3][3] = NO_DAMAGE;
 battle_matrix[0][3][4] = NO_DAMAGE;
 battle_matrix[0][3][5] = ATTACKER_DAMAGE;

 //0 to 4
 battle_matrix[0][4][0] = NO_DAMAGE;
 battle_matrix[0][4][1] = NO_DAMAGE;
 battle_matrix[0][4][2] = NO_DAMAGE;
 battle_matrix[0][4][3] = NO_DAMAGE;
 battle_matrix[0][4][4] = NO_DAMAGE;
 battle_matrix[0][4][5] = ATTACKER_DAMAGE;

 for(i=5;i<12;i++)
  for(j=0;j<6;j++)
   battle_matrix[0][i][j] = NO_DAMAGE;

 //1 to 0
 battle_matrix[1][0][0] = HALF_DAMAGE;
 battle_matrix[1][0][1] = BOTH_DAMAGE;
 battle_matrix[1][0][2] = ATTACKER_DAMAGE;
 battle_matrix[1][0][3] = HALF_DAMAGE;
 battle_matrix[1][0][4] = NO_DAMAGE;
 battle_matrix[1][0][5] = HALF_DAMAGE;

 //1 to 1
  //below

 //1 to 2
 battle_matrix[1][2][0] = HALF_DAMAGE;
 battle_matrix[1][2][1] = BOTH_DAMAGE;
 battle_matrix[1][2][2] = ATTACKER_DAMAGE;
 battle_matrix[1][2][3] = HALF_DAMAGE;
 battle_matrix[1][2][4] = NO_DAMAGE;
 battle_matrix[1][2][5] = HALF_DAMAGE;



 for(i=0;i<12;i++)
 {
  battle_matrix[i][i][0] = HALF_DAMAGE;
  battle_matrix[i][i][1] = BOTH_DAMAGE;
  battle_matrix[i][i][2] = HALF_DAMAGE;
  battle_matrix[i][i][3] = NO_DAMAGE;
  battle_matrix[i][i][4] = NO_DAMAGE;
  battle_matrix[i][i][5] = NO_DAMAGE;
 }
}


CBattle::get_battle_result(int offense, int defense)
{
 srand((unsigned) time(NULL));
 return battle_matrix[offense][defense][rand() % 6];
}
```

sounds.rc

```
#ifdef APSTUDIO_INVOKED
#error : Do not edit this file with AppStudio
#endif

ADD_ARMY_SOUND        sound    test.wav
```

netwar2.rc

```
//Microsoft AppStudio generated resource script.
//
```

```
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////////
/
#undef APSTUDIO_READONLY_SYMBOLS

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""res\\netwar2.rc2"" // non-AppStudio edited resources\r\n"
    "#include ""sounds.rc"" //non-AppStudio edited resources\r\n"
    "#include ""afxres.rc"" \011// Standard components\r\n"
    "\0"
END

/////////////////////////////////////////////////////////////////////////////////
/
#endif // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

IDR_MAINFRAME ICON DISCARDABLE "RES\\NETWAR2.ICO"

/////////////////////////////////////////////////////////////////////////////
//
```

```
// Menu
//

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
       MENUITEM "New Game",                ID_FILE_NEW
       MENUITEM "&Open...\tCtrl+O",        ID_FILE_OPEN
       MENUITEM "&Save\tCtrl+S",           ID_FILE_SAVE
       MENUITEM "Save &As...",             ID_FILE_SAVE_AS
       MENUITEM SEPARATOR
       MENUITEM "Recent File",             ID_FILE_MRU_FILE1, GRAYED
       MENUITEM SEPARATOR
       MENUITEM "E&xit",                   ID_APP_EXIT
    END
    POPUP "Players"
    BEGIN
       MENUITEM "Buy Equipment",           ID_PLAYERS_BUYEQUIPMENT
       MENUITEM "Input Players",           ID_PLAYERS_INPUTPLAYERS
    END
    POPUP "Network"
    BEGIN
       MENUITEM "Winsock - Internet",      ID_NETWORK_WINSOCKINTERNET
       MENUITEM "Serial - Modem",          ID_NETWORK_SERIALMODEM
       MENUITEM "Windows for Workgroups",  ID_NETWORK_WINDOWSFORWORKGROUPS

    END
    MENUITEM "Wait For Connect",        ID_WAITFORCONNECT
    MENUITEM "Connect To Server",       ID_CONNECTTOSERVER
    POPUP "Options"
    BEGIN
       MENUITEM "Sound",                   ID_OPTIONS_SOUND
       MENUITEM "Extra Invalidate",        ID_OPTIONS_EXTRAINVALIDATE
       MENUITEM "Acknowledgements",        ID_OPTIONS_ACKNOWLEDGEMENTS
    END
    POPUP "&Help"
    BEGIN
       MENUITEM "&About Netwar2...",       ID_APP_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
 "N",        ID_FILE_NEW,        VIRTKEY,CONTROL
 "O",          ID_FILE_OPEN,       VIRTKEY,CONTROL
 "S",          ID_FILE_SAVE,       VIRTKEY,CONTROL
 "Z",          ID_EDIT_UNDO,       VIRTKEY,CONTROL
 "X",          ID_EDIT_CUT,        VIRTKEY,CONTROL
 "C",          ID_EDIT_COPY,       VIRTKEY,CONTROL
 "V",          ID_EDIT_PASTE,      VIRTKEY,CONTROL
 VK_BACK,      ID_EDIT_UNDO,       VIRTKEY,ALT
 VK_DELETE,    ID_EDIT_CUT,        VIRTKEY,SHIFT
 VK_INSERT,    ID_EDIT_COPY,       VIRTKEY,CONTROL
 VK_INSERT,    ID_EDIT_PASTE,      VIRTKEY,SHIFT
 VK_F6,        ID_NEXT_PANE,       VIRTKEY
 VK_F6,        ID_PREV_PANE,       VIRTKEY,SHIFT
END


/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE 34, 22, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About Netwar2"
FONT 8, "MS Sans Serif"
BEGIN
 ICON           IDR_MAINFRAME,IDC_STATIC,11,17,20,20
 LTEXT          "Netwar2 Application Version 1.0",IDC_STATIC,40,10,119,8
 LTEXT          "Copyright \251 1995",IDC_STATIC,40,25,119,8
 DEFPUSHBUTTON  "OK",IDOK,176,6,32,14,WS_GROUP
END

IDD_PLAYING_PIECE_STATS DIALOG DISCARDABLE 0, 0, 187, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Playing Piece Statistics"
FONT 8, "MS Sans Serif"
BEGIN
 DEFPUSHBUTTON  "OK",IDOK,68,72,50,14
 LTEXT          "Type:",IDC_STATS_TYPE,8,13,68,8
 LTEXT          "Offense:",IDC_STATS_OFFENSE,8,40,68,8
 LTEXT          "Owner:",IDC_STATS_OWNER,92,16,68,8
 LTEXT          "Defense:",IDC_STATS_DEFENSE,92,40,68,8
END

IDD_OPERATIONS_DIALOG DIALOG DISCARDABLE 510, 0, 74, 277
STYLE DS_ABSALIGN | DS_MODALFRAME | WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE |
 WS_CAPTION | WS_SYSMENU
CAPTION " "
FONT 8, "MS Sans Serif"
BEGIN
 LTEXT          "Armies",IDC_STATIC,4,4,23,9
 LISTBOX        IDC_AVAILABLE_ARMIES,4,16,64,60,LBS_NOINTEGRALHEIGHT |
                WS VSCROLL | WS TABSTOP
 LTEXT          "Gold",Gold,4,85,16,8
 EDITTEXT       IDC_OPERATIONS AVAILABLE_GOLD,4,100,68,12,ES_AUTOHSCROLL |
                ES_READONLY | NOT WS_BORDER
 EDITTEXT       IDC OPERATIONS WAR_STATEMENT,2,120,38,13,ES AUTOHSCROLL
 PUSHBUTTON     "WAR",IDC WAR_BUTTON,44,120,24,11
 PUSHBUTTON     "I Quit",IDC OPERATIONS_I_QUIT,24,140,25,11
 PUSHBUTTON     "Message",IDC_OPERATIONS MESSAGE_BUTTON,13,157,46,10
 LISTBOX        IDC_OPERATIONS_MESSAGE BOX,3,169,69,103,NOT LBS_NOTIFY |
                WS_VSCROLL | WS HSCROLL | WS_TABSTOP
END

IDD BUY EQUIPMENT DIALOG DISCARDABLE 0, 0, 358, 266
STYLE DS_MODALFRAME | WS POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Buy Armies"
FONT 8, "MS Sans Serif"
BEGIN
 DEFPUSHBUTTON  "OK",IDOK,106,248,50,14
 PUSHBUTTON     "Cancel",IDCANCEL,202,248,50,14
 LTEXT          "Available Gold Coins:",IDC_STATIC,4,12,72,8
 LTEXT          "Armies:",IDC STATIC,15,32,25,8
 ICON           126,IDC STATIC,55,44,18,20
 LTEXT          "Cavalry",IDC_STATIC,19,48,31,16
 LTEXT          "Offense = 4; Defense = 3; Cost = 20,000 coins; Speed = 4",
                IDC STATIC,79,52,208,8
 LTEXT          "Archers",IDC_STATIC,19,104,33,8
 ICON           125,IDC STATIC,55,92,18,20
 LTEXT          "Offense = 4; Defense = 1; Cost = 10,000 coins; Speed = 3",
                IDC STATIC,79,100,207,8
 PUSHBUTTON     "Buy Cavalry",IDC_BUY CAVALRYS,91,68,49,12
 PUSHBUTTON     "Buy Archers",IDC_BUY ARCHERS,91,116,49,12
 PUSHBUTTON     "Sell Archers",IDC SELL_ARCHERS,163,116,49,12
 PUSHBUTTON     "Sell Cavalry",IDC_SELL_CAVALRYS,163,68,49,12
 LTEXT          "Bought =",IDC STATIC,295,56,32,8
 LTEXT          "Bought =",IDC STATIC,295,100,32,8
 EDITTEXT       IDC BUY CAVALRYS COUNT,332,55,11,14,ES_READONLY | NOT
                WS BORDER
 EDITTEXT       IDC BUY ARCHERS_COUNT,335,96,8,16,ES READONLY | NOT
                WS BORDER
 EDITTEXT       IDC BUY AVAILABLE COINS,84,13,49,11,ES_READONLY | NOT
                WS BORDER
 LTEXT          "Ground Legion",IDC_STATIC,19,140,28,18
 ICON           127,IDC_STATIC,55,136,18,20
 LTEXT          "Offense = 8; Defense = 6; Cost = 20,000 coins; Speed = 2",
                IDC STATIC,79,144,195,10
 LTEXT          "Flyers",IDC STATIC,19,184,23,10
 ICON           128,IDC STATIC,55,180,18,20
 LTEXT          "Offense = 3; Defense = 3; Cost = 40,000 coins; Speed = 10",
                IDC STATIC,79,184,176,13
 PUSHBUTTON     "Buy Legion",IDC_BUY LEGION,91,160,49,12
 PUSHBUTTON     "Sell Legion",IDC_SELL LEGION,163,160,49,12
```

```
  PUSHBUTTON      "Sell Flyers",IDC_SELL_FLYERS,163,204,49,12
  PUSHBUTTON      "Buy Flyers",IDC_BUY_FLYERS,91,204,49,12
  LTEXT           "Bought = ",IDC_STATIC,295,140,32,10
  LTEXT           "Bought =",IDC_STATIC,293,183,30,9
  EDITTEXT IDC_BUY_LEGIONS_COUNT,332,137,11,15,ES_READONLY | NOT
                  WS_BORDER
  EDITTEXT IDC_BUY_FLYERS_COUNT,331,180,10,13,ES_READONLY | NOT
                  WS_BORDER
END

IDD_INPUT_SELF DIALOG DISCARDABLE 0, 0, 187, 89
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Input Information about Yourself"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON   "OK",ID_OK,64,68,50,14
  CONTROL         "Client",IDC_RADIO_CLIENT,"Button",BS_AUTORADIOBUTTON,43,
                  12,28,8
  CONTROL         "Server",IDC_RADIO_SERVER,"Button",BS_AUTORADIOBUTTON,
                  113,12,30,10
  LTEXT           "Total Players = ",-1,53,41,54,8
  EDITTEXT        IDC_INPUT_TOTAL_PLAYERS,112,38,20,12,ES_AUTOHSCROLL
END

IDD_INPUT_SERVER DIALOG DISCARDABLE 0, 0, 191, 77
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Input Information about Server"
FONT 8, "MS Sans Serif"
BEGIN
  LTEXT           "Internet Address:",IDC_EDIT2,8,12,60,8
  EDITTEXT        IDC_INPUT_PLAYER_INTERNET,68,12,115,12,ES_AUTOHSCROLL
  PUSHBUTTON      "OK",ID_OK,33,52,50,14
  PUSHBUTTON      "Cancel",IDCANCEL,103,52,50,14
  LTEXT           "or Telephone #",-1,8,22,51,8
END

IDD_SERVER_WAIT DIALOG DISCARDABLE 0, 0, 187, 41
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Server Message"
FONT 8, "MS Sans Serif"
BEGIN
  LTEXT           "Server Waiting for Connections from Clients",IDC_STATIC,
                  21,16,143,9
END

IDD_MESSAGE_INPUT DIALOG DISCARDABLE 0, 0, 254, 93
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Send Message"
FONT 8, "MS Sans Serif"
BEGIN
  PUSHBUTTON      "Cancel",IDCANCEL,169,72,50,12
```

```
 EDITTEXT       IDC_MESSAGE_INPUT,66,36,121,14
 LTEXT          "Enter your message and then click on the appropriate button",
                IDC_STATIC,26,16,200,9
 PUSHBUTTON     "0",IDC_SEND_MESSAGE_1,34,72,16,11
 PUSHBUTTON     "1",IDC_SEND_MESSAGE_2,61,72,16,11
 PUSHBUTTON     "2",IDC_SEND_MESSAGE_3,89,72,16,11
 PUSHBUTTON     "3",IDC_SEND_MESSAGE_4,117,72,16,11
 PUSHBUTTON     "ALL",IDC_SEND_MESSAGE_ALL,142,72,16,11
END

IDD_COMM_INPUT DIALOG DISCARDABLE 0, 0, 188, 112
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Communication Information"
FONT 8, "MS Sans Serif"
BEGIN
 DEFPUSHBUTTON  "OK",IDOK,35,95,50,14
 PUSHBUTTON     "Cancel",IDCANCEL,100,95,50,14
 GROUPBOX       "Com Port",IDC_STATIC,26,5,42,55,WS_GROUP
 CONTROL        "COM 1",IDC_COM_1,"Button",BS_AUTORADIOBUTTON,31,15,33,
                10
 CONTROL        "COM 2",IDC_COM_2,"Button",BS_AUTORADIOBUTTON,31,25,33,8
 CONTROL        "COM 3",IDC_COM_3,"Button",BS_AUTORADIOBUTTON,31,35,33,8
 CONTROL        "COM 4",IDC_COM_4,"Button",BS_AUTORADIOBUTTON,31,45,33,8
 GROUPBOX       "Baud Rate",IDC_STATIC,80,8,84,45,WS_GROUP
 CONTROL        "2400",IDC_BAUD_2400,"Button",BS_AUTORADIOBUTTON,91,20,
                25,8
 CONTROL        "4800",IDC_BAUD_4800,"Button",BS_AUTORADIOBUTTON,91,30,
                25,8
 CONTROL        "9600",IDC_BAUD_9600,"Button",BS_AUTORADIOBUTTON,91,40,
                25,8
 CONTROL        "19200",IDC_BAUD_19200,"Button",BS_AUTORADIOBUTTON,126,
                20,30,8
 CONTROL        "38400",IDC_BAUD_38400,"Button",BS_AUTORADIOBUTTON,126,
                30,30,8
 CONTROL        "57600",IDC_BAUD_57600,"Button",BS_AUTORADIOBUTTON,126,
                40,30,8
 LTEXT          "Modem Initialization String",IDC_STATIC,8,64,48,28
 EDITTEXT       IDC_INPUT_INITIALIZATION_STRING,65,75,119,12,
                ES_AUTOHSCROLL
END


/////////////////////////////////////////////////////////////////////////////
//
// Bitmap
//

IDB_PLAYING_FIELD          BITMAP DISCARDABLE    "BACKGRND.BMP"
IDB_PO_ARCHER_BITMAP       BITMAP DISCARDABLE    "RES\\NEW_ARCH.BMP"
IDB_PO_ARCHER_HL           BITMAP DISCARDABLE    "RES\\BMP00066.BMP"
```

```
IDB_P0_CAVALRY_BITMAP       BITMAP DISCARDABLE      "RES\\NEW_CAVA.BMP"
IDB_P0_CAVALRY_HL           BITMAP DISCARDABLE      "RES\\BMP00065.BMP"
IDB_P0_FLYER_BITMAP         BITMAP DISCARDABLE      "RES\\NEW_FLYE.BMP"
IDB_P0_FLYER_HL             BITMAP DISCARDABLE      "RES\\BMP00064.BMP"
IDB_P0_FLYER_MASK           BITMAP DISCARDABLE      "RES\\BMP00063.BMP"
IDB_P0_LEGION_BITMAP        BITMAP DISCARDABLE      "RES\\NEW_LEGI.BMP"
IDB_P0_LEGION_HL            BITMAP DISCARDABLE      "RES\\BMP00067.BMP"
IDB_P0_ARCHER_MASK          BITMAP DISCARDABLE      "RES\\P0_ARCHE.BMP"
IDB_P0_CAVALRY_MASK         BITMAP DISCARDABLE      "RES\\P0_CAVAL.BMP"
IDB_P0_LEGION_MASK          BITMAP DISCARDABLE      "RES\\P0_LEGIO.BMP"
IDB_P1_ARCHER_BITMAP        BITMAP DISCARDABLE      "RES\\P1_ARCHE.BMP"
IDB_P1_ARCHER_HL            BITMAP DISCARDABLE      "RES\\BMP00001.BMP"
IDB_P1_ARCHER_MASK          BITMAP DISCARDABLE      "RES\\BMP00002.BMP"
IDB_P1_CAVALRY_BITMAP       BITMAP DISCARDABLE      "RES\\P1_CAVAL.BMP"
IDB_P1_CAVALRY_HL           BITMAP DISCARDABLE      "RES\\BMP00003.BMP"
IDB_P1_CAVALRY_MASK         BITMAP DISCARDABLE      "RES\\BMP00004.BMP"
IDB_P1_FLYER_BITMAP         BITMAP DISCARDABLE      "RES\\P1_FLYER.BMP"
IDB_P1_FLYER_HL             BITMAP DISCARDABLE      "RES\\BMP00006.BMP"
IDB_P1_FLYER_MASK           BITMAP DISCARDABLE      "RES\\BMP00005.BMP"
IDB_P1_LEGION_BITMAP        BITMAP DISCARDABLE      "RES\\P1_LEGIO.BMP"
IDB_P1_LEGION_HL            BITMAP DISCARDABLE      "RES\\BMP00007.BMP"
IDB_P1_LEGION_MASK          BITMAP DISCARDABLE      "RES\\BMP00009.BMP"
IDB_P2_ARCHER_BITMAP        BITMAP DISCARDABLE      "RES\\P2_ARCHE.BMP"
IDB_P2_ARCHER_HL            BITMAP DISCARDABLE      "RES\\BMP00008.BMP"
IDB_P2_ARCHER_MASK          BITMAP DISCARDABLE      "RES\\BMP00010.BMP"
IDB_P2_CAVALRY_BITMAP       BITMAP DISCARDABLE      "RES\\P2_CAVAL.BMP"
IDB_P2_CAVALRY_HL           BITMAP DISCARDABLE      "RES\\BMP00011.BMP"
IDB_P2_CAVALRY_MASK         BITMAP DISCARDABLE      "RES\\BMP00012.BMP"
IDB_P2_FLYER_BITMAP         BITMAP DISCARDABLE      "RES\\P2_FLYER.BMP"
IDB_P2_FLYER_HL             BITMAP DISCARDABLE      "RES\\BMP00013.BMP"
IDB_P2_FLYER_MASK           BITMAP DISCARDABLE      "RES\\BMP00014.BMP"
IDB_P2_LEGION_BITMAP        BITMAP DISCARDABLE      "RES\\P2_LEGIO.BMP"
IDB_P2_LEGION_HL            BITMAP DISCARDABLE      "RES\\BMP00015.BMP"
IDB_P2_LEGION_MASK          BITMAP DISCARDABLE      "RES\\BMP00016.BMP"
IDB_P3_ARCHER_BITMAP        BITMAP DISCARDABLE      "RES\\P3_ARCHE.BMP"
IDB_P3_ARCHER_HL            BITMAP DISCARDABLE      "RES\\BMP00024.BMP"
IDB_P3_ARCHER_MASK          BITMAP DISCARDABLE      "RES\\BMP00023.BMP"
IDB_P3_CAVALRY_BITMAP       BITMAP DISCARDABLE      "RES\\BMP00022.BMP"
IDB_P3_CAVALRY_HL           BITMAP DISCARDABLE      "RES\\BMP00021.BMP"
IDB_P3_CAVALRY_MASK         BITMAP DISCARDABLE      "RES\\P3_CAVAL.BMP"
IDB_P3_FLYER_BITMAP         BITMAP DISCARDABLE      "RES\\BMP00020.BMP"
IDB_P3_FLYER_HL             BITMAP DISCARDABLE      "RES\\BMP00019.BMP"
IDB_P3_FLYER_MASK           BITMAP DISCARDABLE      "RES\\P3_FLYER.BMP"
IDB_P3_LEGION_BITMAP        BITMAP DISCARDABLE      "RES\\BMP00018.BMP"
IDB_P3_LEGION_HL            BITMAP DISCARDABLE      "RES\\BMP00017.BMP"
IDB_P3_LEGION_MASK          BITMAP DISCARDABLE      "RES\\P3_LEGIO.BMP"

/////////////////////////////////////////////////////////////////////////////
//
// String Table
//
```

```
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME          "Netwar2 Windows Application\nNetwar\nNetwar
Document\n\n\nNetwar.Document\nNetwar Document"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  AFX_IDS_APP_TITLE      "Netwar2 Windows Application"
  AFX_IDS_IDLEMESSAGE    "Ready"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_INDICATOR_EXT       "EXT"
  ID_INDICATOR_CAPS      "CAP"
  ID_INDICATOR_NUM       "NUM"
  ID_INDICATOR_SCRL      "SCRL"
  ID_INDICATOR_OVR       "OVR"
  ID_INDICATOR_REC       "REC"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_NEW            "Create a new document"
  ID_FILE_OPEN           "Open an existing document"
  ID_FILE_CLOSE          "Close the active document"
  ID_FILE_SAVE           "Save the active document"
  ID_FILE_SAVE_AS        "Save the active document with a new name"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_APP_ABOUT           "Display program information, version number and copyright"
  ID_APP_EXIT            "Quit the application; prompt to save documents"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_MRU_FILE1      "Open this document"
  ID_FILE_MRU_FILE2      "Open this document"
  ID_FILE_MRU_FILE3      "Open this document"
  ID_FILE_MRU_FILE4      "Open this document"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_NEXT_PANE           "Switch to the next window pane"
  ID_PREV_PANE           "Switch back to the previous window pane"
END
```

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_EDIT_CLEAR         "Erase the selection"
  ID_EDIT_CLEAR_ALL     "Erase everything"
  ID_EDIT_COPY          "Copy the selection and put it on the Clipboard"
  ID_EDIT_CUT           "Cut the selection and put it on the Clipboard"
  ID_EDIT_FIND          "Find the specified text"
  ID_EDIT_PASTE         "Insert Clipboard contents"
  ID_EDIT_REPEAT        "Repeat the last action"
  ID_EDIT_REPLACE       "Replace specific text with different text"
  ID_EDIT_SELECT_ALL    "Select the entire document"
  ID_EDIT_UNDO          "Undo the last action"
  ID_EDIT_REDO          "Redo the previously undone action"
END

STRINGTABLE DISCARDABLE
BEGIN
  AFX_IDS_SCSIZE        "Change the window size"
  AFX_IDS_SCMOVE        "Change the window position"
  AFX_IDS_SCMINIMIZE    "Reduce the window to an icon"
  AFX_IDS_SCMAXIMIZE    "Enlarge the window to full size"
  AFX_IDS_SCNEXTWINDOW  "Switch to the next document window"
  AFX_IDS_SCPREVWINDOW  "Switch to the previous document window"
  AFX_IDS_SCCLOSE       "Close the active window and prompt to save the documents"
END

STRINGTABLE DISCARDABLE
BEGIN
  AFX_IDS_SCRESTORE     "Restore the window to normal size"
  AFX_IDS_SCTASKLIST    "Activate Task List"
END


#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
#include "res\netwar2.rc2" // non-App Studio edited resources
#include "sounds.rc" //non-AppStudio edited resources
#include "afxres.rc" // Standard components

//////////////////////////////////////////////////////////////////////////////
/
#endif // not APSTUDIO_INVOKED
```

Appendix C

Enclosed CD-ROM

The enclosed CD-ROM contains both source code and executables for King's Reign, a multiplayer war game for up to four players that works over modems, networks, and the Internet. See Appendix A for details on playing the game.

You'll also get a powerful collection of paint programs, icon developers, sound utilities, Internet utilities, source code, and technical documentation. That will help you design and build your own multiplayer Windows Internet game.

Some of the software on the enclosed CD-ROM for this book is provided as shareware. If you find any of the software useful and choose to keep it, please check the registration requirements set forth by the author of the software. Purchasing this book does not include a payment to the authors of any of the software on the CD-ROM.

Specifically: The Graphics Workshop for Windows software on the companion CD-ROM for this book is provided as shareware. If you find it useful and choose to keep it, please register it as described in its documentation. The purchase price of this book does not include a payment to the author of Graphics Workshop.

The source code for the enclosed game, King's Reign is provided to you the bookbuyer license free. You may use this code in part or in its entirety in any commercial product you develop provided that you credit the author, Joe Gradecki, and mention the title of this book.

INDEX

Games/C++

$34.95 USA
$48.95 CAN

Create sophisticated multiplayer games in C++ for Windows® 3.1 and Windows® 95 that work over modems, ethernet networks, and the Internet

Want to create your own multiplayer network game? Do it with NetWarriors in C++. This powerful book/CD package supplies you with all the know-how and tools you need to create fun, bugfree multiplayer games for Windows® 3.1 and Windows® 95.

Organized around the example of "King's Reign," an exciting game of combat and political intrigue set in medieval times, this book takes you through all the steps of creating the game from the ground up, one line of code at a time. You'll learn object-oriented programming techniques for Visual C++, Windows 3.1 and Windows 95, how to use graphics techniques like sprites, bitmaps, animation, and 3D objects, how to program music and sound effects, and much more.

On the CD-ROM you'll find:

- **"King's Reign," an original, ready-to-run multiplayer game**
- **All the C++ code for "King's Reign"**
- **A gold mine of powerful graphics tools and other programming utilities**
- **Additional programming resources**

JOE GRADECKI is the author of the popular NetWarriors in C, The Virtual Reality Programming Kit, and The Virtual Reality Construction Kit, all from Wiley.

Cover Design: Howard Grossman
Cover Painting: David Miller

WILEY COMPUTER PUBLISHING

John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0-471-11337-9
53495
9 780471 113379